

The multiresolution analysis of flow graphs

Steve Huntsman*

BAE Systems, 4301 North Fairfax Drive, Arlington, Virginia 22203

(Dated: January 26, 2017)

We introduce and prove basic results about several graph-theoretic notions relevant to the multiresolution analysis of *flow graphs*. A categorical viewpoint is taken in later sections to demonstrate that our definitions are natural and to motivate particular incarnations of related constructions. Examples are discussed and code is included in appendices.

I. INTRODUCTION

The notion of a “flow graph” is central to the analysis and compilation of computer programs, encompassing constructs that represent the transfer of control and data [1–3]. As the complexity of software increases, so does the scale of the corresponding flow graphs: accordingly, a framework for the analysis of flow graphs at multiple resolutions is required. Such a framework was originally presented in [4], where a hierarchical representation of input/output structure called the *program structure tree* (PST) was constructed.

This paper extends (and corrects: cf. [5]) the work of [4] by unifying and formalizing several natural concepts relevant to the decomposition and construction of flow graphs. This has several benefits: as the most basic example, we provide a definition of flow graph that is slightly different than its other usual variants but that is mathematically more natural and well-behaved. This in turn leads to a simpler analogue of the “refined process structure tree” of [6, 7] and natural category-theoretic constructions, notably multiresolution and composition operations on flow graphs. While most of the results of this paper are conceptually straightforward and many are at least latent in the literature, few of them have been simultaneously formulated explicitly and mathematically. Indeed, the practical motivation for this paper is simply to show that the “right” definition of a flow graph entails all the obvious desiderata.

The paper is organized as follows: we discuss dominance relations in §II; flow graphs, single-entry/single-exit regions, and the PST in §III; the stretching operation in §IV; we introduce the structure of a category on flow graphs in §V; we discuss multiresolution transformations on flow graphs in §VI; and in §VII we discuss series and parallel composition of flow graphs in the context of formal tensor product structures. Appendices contain a discussion of two-terminal graphs (§A), the proofs of several technical results (§B), computational examples (§C), corresponding code (§D), and supplementary code for dominance relations (§E).

We remark at the outset that all graphs (and related objects) are assumed finite throughout this paper. By convention, digraphs are allowed to have loops from a vertex to itself. Given a vertex v in a digraph, let $d_0^+(v)$, $d_0^-(v)$, and $d^0(v)$ respectively denote the number of incoming edges excluding any loop, the number of outgoing edges excluding any loop, and the number (≤ 1) of loops at v . A vertex v is a *source* iff $d_0^+(v) = 0$ and a *target* iff $d_0^-(v) = 0$, i.e., loops have no bearing on these properties.

II. DOMINANCE RELATIONS

Recall that the number of paths of length ℓ from a vertex j to a vertex k in a digraph G is given by $(A^\ell)_{jk}$, where $A \equiv A(G)$ is the adjacency matrix of G . Also, recall that A is nilpotent iff G is acyclic. In this event, it is easy to prove that the number N_{jk} of paths from j to k of any length is finite and given by

$$N_{jk} = \left(\sum_{\ell=0}^{\infty} A^\ell\right)_{jk} = ([I - A]^{-1})_{jk}. \quad (1)$$

In particular, if A is the adjacency matrix of a directed tree, then j is an ancestor of k iff $(N - I)_{jk} = 1$, and a descendant of k iff $(N - I)_{kj} = 1$.

We say that j *dominates* k , written $j \text{ dom } k$, iff every path from a source s in G to k passes through j [2, 3]. The dominance relation extends to edges without any further comment here. While in light of (1) it is easy to characterize the dominance relation algebraically, viz. (under mild assumptions)

$$j \text{ dom } k \Leftrightarrow (A^{m+n})_{sk} = (A^m)_{sj} \cdot (A^n)_{jk}, \quad \forall m, n; s \text{ a source in } G, \quad (2)$$

*Electronic address: steve.huntsman@baesystems.com

this characterization is only convenient to work with in the case of DAGs, where it simplifies to

$$j \text{ dom } k \Leftrightarrow N_{sk} = N_{sj} \cdot N_{jk}, \quad \forall s \text{ a source in } G. \quad (3)$$

While this leads to a dominance algorithm for (nice, e.g., single-source) DAGs that can be implemented in just a few lines of MATLAB, [32] the generally preferred avenues for computing the dominance relation [8–11] have a very different flavor typically emphasizing depth-first search or iteration in lattices.

Define

$$D_{jk} := \begin{cases} 1 & \text{if } j \text{ dom } k \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

Similarly, let $D^\dagger := D(G^*)$, where G^* is the reversal or adjoint of G with adjacency matrix A^* and corresponding dominance relation dom^\dagger . If $D_{jk}^\dagger = 1$, i.e., if $j \text{ dom}^\dagger k$, write $k \text{ pdom } j$ and say that k *postdominates* j . Like dominance, the postdominance relation extends to edges.

PROPOSITION. For distinct edges $\{e_j\}_{j=1}^3$ in a digraph G , if $e_1 \text{ dom } e_3$ and $e_2 \text{ dom } e_3$, then either $e_1 \text{ dom } e_2$ or $e_2 \text{ dom } e_1$. Similarly, if $e_1 \text{ pdom } e_2$ and $e_1 \text{ pdom } e_3$, then either $e_2 \text{ pdom } e_3$ or $e_3 \text{ pdom } e_2$. \square

PROPOSITION. If $e_1 \text{ dom } e_2$ and $e_1 \text{ pdom } e_2$ with $e_1 \neq e_2$, then any path from a source to a target that traverses e_2 contains a cycle of the form $(e_1, \dots, e_2, \dots, e_1)$. \square

The immediately preceding proposition is important for addressing a subtle error in [4]; cf. [5].

III. FLOW GRAPHS, SINGLE-ENTRY/SINGLE-EXIT REGIONS, AND THE PROGRAM STRUCTURE TREE

If G is a digraph, let an *encapsulation* of G be a minimal digraph containing G , having exactly one source and exactly one target, and such that there is a unique (entry) edge from the source and a unique (exit) edge to the target. (We do not require the entry and exit edges to be distinct in the case that G has two or fewer vertices.)

PROPOSITION. Let G be a digraph with at least one source and at least one target. Then there is a unique encapsulation of G . \square

While the proposition is obvious, the detailed construction of the encapsulation *in silico* turns out to be perhaps more delicate than expected (cf. §D 2). Note also that the class of digraphs considered in the proposition is essentially the largest possible: if a nonempty digraph does not have any source vertex, then it necessarily has more than one vertex, and therefore has more than one encapsulation; a similar observation applies for digraphs with no target vertices. In such events we may, when convenient, use the convention that “the” encapsulation is specified in terms of an ordering on the vertices by introducing a source edge to the first vertex and/or a target edge from the last vertex as necessary: the code in §D 2 follows this convention.

A digraph that is its own encapsulation and has a path cover consisting solely of paths from the source to the target is called a *flow graph*. [33]

PROPOSITION. Every edge of a flow graph is on some path from the source to the target. \square

A *single entry/single exit (SESE) region* in a digraph G is defined as an ordered pair of edges (e_1, e_2) satisfying each of the following conditions [4]:

- $e_1 \text{ dom } e_2$;
- $e_2 \text{ pdom } e_1$;
- a cycle in G contains e_1 iff it contains e_2 .

See figure 1. Note first that (e_1, e_1) is a degenerate SESE region [34], and second that a nondegenerate SESE region (e_1, e_2) (i.e., a SESE region with $e_1 \neq e_2$) unambiguously corresponds to the ordered vertex pair $(t(e_1), s(e_2))$, where $s(\cdot)$ and $t(\cdot)$ respectively denote the source and target of an edge. We may use either the edge or vertex pairs above to specify a nondegenerate SESE region. Note also that in a DAG the third condition above is trivial. Finally, note

that the edges e_s from the source and e_t to the target of a flow graph G together define a SESE region and *vice versa*. With this in mind, write either G or (e_s, e_t) for the flow graph or the equivalent SESE region.

We give a few simple results before moving on to a fundamental theorem.

PROPOSITION. If (e_1, e_2) and (e_2, e_3) are SESE regions, then so is (e_1, e_3) . \square

LEMMA. If (e_1, e_2) and (e_1, e_3) are SESE regions with $e_2 \neq e_3$ and $e_2 \text{ dom } e_3$, then (e_2, e_3) is a (nondegenerate) SESE region.

PROOF. The only thing to show is that $e_3 \text{ pdom } e_2$. It must be the case that either $e_2 \text{ pdom } e_3$ or $e_3 \text{ pdom } e_2$, so assume the former. Since $e_2 \text{ dom } e_3$ also, we must have that any source-target path traversing e_3 contains a cycle of the form $(e_2, \dots, e_3, \dots, e_2)$; deleting all cycles from this path yields a source-target path traversing e_2 but not e_3 . Reversing this path yields a contradiction to the assumption that $e_2 \text{ pdom } e_3$. \square

COROLLARY. If (e_1, e_2) is a SESE region with $e_2 \neq e_3$ and $e_2 \text{ dom } e_3$, and (e_2, e_3) is not a SESE region, then (e_1, e_3) is also not a SESE region. [35] \square

The *boundary* of $G \equiv (e_s, e_t)$ is $\partial G := \{s(e_s), t(e_t)\}$ and the *interior* G° of G is the set of vertices on paths starting from $t(e_s)$ that do not encounter $t(e_t)$. [36] A nondegenerate SESE region (e_1, e_2) is called *canonical* if for any SESE region (e_1, e'_2) it is the case that $e_2 \text{ dom } e'_2$ and if for any SESE region (e'_1, e_2) it is the case that $e_1 \text{ pdom } e'_1$. Our definition of the interior of a SESE region enables the following corrected version of theorem 1 of [4] (cf. [5]):

THEOREM. The interiors of distinct canonical SESE regions are either disjoint or nested.

PROOF. In §B 1. \square

Therefore canonical SESE regions are also *minimal* in the corresponding obvious sense, so we may use the two terms interchangeably (we generally prefer and use the latter). The inclusion relation on minimal SESE regions induces a tree—viz., the PST. An example of this nesting behavior and the corresponding PST are depicted in figures 1 and 2, respectively.

LEMMA. A nondegenerate SESE region (e_0, e_∞) can be decomposed as $(e_0, e_\infty) = \bigcup_{j=1}^m (e_{j-1}, e_j)$, where $e_m \equiv e_\infty$ and (e_{j-1}, e_j) is a minimal SESE region for $1 \leq j \leq m$.

PROOF. Suppose w.l.o.g. that (e_0, e_∞) is not minimal. Then at least one of the following is true: i) there exists a nondegenerate SESE region (e_0, e_1) such that e_∞ does not dominate e_1 ; ii) there exists a nondegenerate SESE region (e_{-1}, e_∞) such that e_0 does not postdominate e_{-1} . Consider case i), and assume w.l.o.g. that (e_0, e_1) is minimal (otherwise, we have at least one of case i) or ii) again). Then $e_1 \text{ dom } e_\infty$, so (e_1, e_∞) is a nondegenerate SESE region and we can write $(e_0, e_\infty) = (e_0, e_1) \cup (e_1, e_\infty)$. Exactly similar reasoning informs case ii), and an induction establishes the lemma. \square

Define a matrix S indexed by edges by $S_{e_1, e_2} = 1$ if (e_1, e_2) is a nondegenerate SESE region and $S_{e_1, e_2} = 0$ otherwise. Then S is the adjacency matrix of a digraph whose weakly connected components correspond to the situation in the preceding lemma. We therefore obtain the following

PROPOSITION. Each weakly connected component of the digraph corresponding to S is a transitive tournament, hence has a unique source and a unique target, and a unique path of length 1 from the source to the target which defines a *locally maximal SESE region*. \square

A closely related construction is the subject of §VI.

IV. STRETCHING FLOW GRAPHS

In practice, we may be interested in subgraphs that are only “approximately” flow graphs, and therefore not captured by the program structure tree of the original graph. For instance if a digraph H is isomorphic to the result of encapsulating H and then discarding the entry and exit edges along with the source and target, then it is not a flow graph: nevertheless, if $H \subset G$, this may still be of interest to us. By inserting new vertices and edges in G based

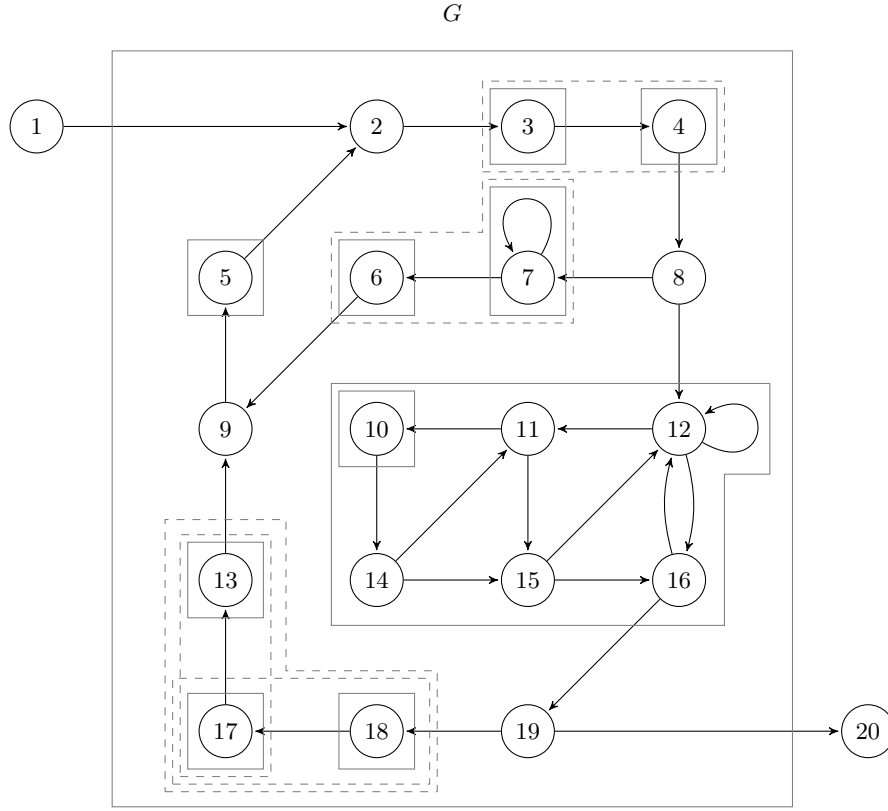


FIG. 1: SESE regions of the flow graph G are outlined in gray: minimal SESE region outlines are solid, while non-minimal SESE region outlines are dashed. The locally maximal SESE regions that are not also minimal SESE regions are $((2, 3), (4, 8))$, $((8, 7), (6, 9))$, and $((19, 18), (13, 9))$.

on the presence of multiple incoming and outgoing edges (or loops), such subgraphs can be transformed into sub-flow graphs, as the following lemma details.

LEMMA. Let G be a flow graph. For each vertex $v \in G^\circ$, perform the following transformations on G :

- If $d_0^+(v) = 1$, $d_0^-(v) > 1$, and $d^0(v) = 1$, then replace v with two vertices v_s and v_t , reassign both the target of the original incoming non-loop edge and the original loop to v_s , reassign the source of the original outgoing non-loop edges to v_t , and finally introduce an edge from v_s to v_t .
- If $d_0^+(v) > 1$, $d_0^-(v) = 1$, and $d^0(v) = 1$, then replace v with two vertices v_s and v_t , reassign the target of the original incoming non-loop edges to v_s , reassign both the source of the original outgoing non-loop edge and the loop to v_t , and finally introduce an edge from v_s to v_t .
- If $d_0^+(v) > 1$, $d_0^-(v) > 1$, and $d^0(v) = 0$, then replace v with two vertices v_s and v_t , reassign the target of the original incoming edges to v_s , reassign the source of the original outgoing edges to v_t , and finally introduce an edge from v_s to v_t .
- If $d_0^+(v) > 1$, $d_0^-(v) > 1$, and $d^0(v) = 1$, then replace v with three vertices v_s , v' , and v_t , reassign the target of the original incoming non-loop edges to v_s , reassign the loop to v' , reassign the source of the original outgoing non-loop edges to v_t , and finally introduce edges from v_s to v' and from v' to v_t .

Then the cumulative result of these transformations is well-defined (i.e., it does not depend on the order in which the transformations are performed) and stable, in the sense that none of the preceding cases applies to any of the vertices in the resulting graph. That is, repeating these transformations has no effect.

PROOF. A diagrammatic proof is transparent and also illuminates the statement of the lemma itself. To wit, the following table outlines all eight possibilities for an interior vertex of a flow graph, showing new edges introduced by a transformation as dashed in the last column:

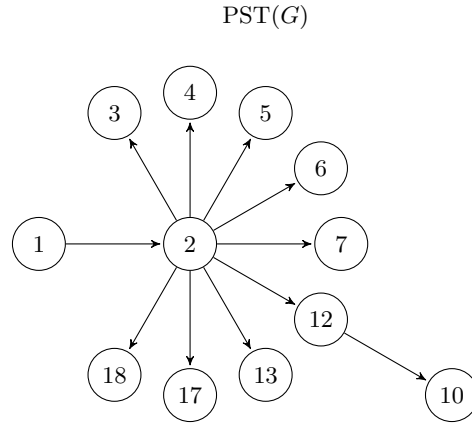


FIG. 2: The program structure tree (PST) encodes the nesting of minimal SESE regions. Here, nodes are labeled by the target of the incoming edge (for the sake of convention, there are “phantom” edges from $-\infty$ to the source of a flow graph and from the target of a flow graph to ∞). The sets $\{3, 4\}$, $\{6, 7\}$, and $\{13, 17, 18\}$ correspond to locally maximal SESE regions that could sensibly be “aggregated” by identifying the respective vertices and annihilating any resulting loops: however, a more mathematically natural variant of this construction is discussed in §VI.

$d^+ > 1?$	$d^- > 1?$	$d^0 = 1?$	lemma case	original motif	new motif
no	no	no	-		
no	no	yes	-		
no	yes	no	-		
no	yes	yes	first		
yes	no	no	-		
yes	no	yes	second		
yes	yes	no	third		
yes	yes	yes	fourth		

In each of the new motifs, none of the four original cases enumerated in the lemma’s statement applies to any of the internal vertices. The lemma follows. \square

Call the cumulative result of the transformations in the lemma above the *stretching* of G (cf. figures 3 and 4). This construction is directly analogous to the “normalization” of two-terminal graphs (see §A).

COROLLARY. There is a bijective correspondence between induced subgraphs of G with single sources and targets and SESE regions in the stretching of G . In particular, any loop in G corresponds to a (necessarily minimal) SESE region in the stretching of G . \square

We conclude this section by noting one way that stretching can induce complexity in the structure of a flow graph. While the particular stretching depicted in figures 3 and 4 turns out to preserve planarity, in general this will not be the case, as the following lemma illustrated in figure 5 shows.

LEMMA. There exists a planar flow graph whose stretching is nonplanar.

PROOF. Consider the planar flow graph depicted on the left portion of figure 5 (to see planarity, move vertex 6 into the square formed by vertices 2-5). Stretching only affects vertex 6, which is split into two vertices 6_s and 6_t as shown in the right portion of the figure. The undirected graph corresponding to the stretching contains $K_{3,3}$ (the complete bipartite graph on three “square” and three “diamond” vertices) as a subgraph, and hence is nonplanar by Kuratowski’s theorem. \square

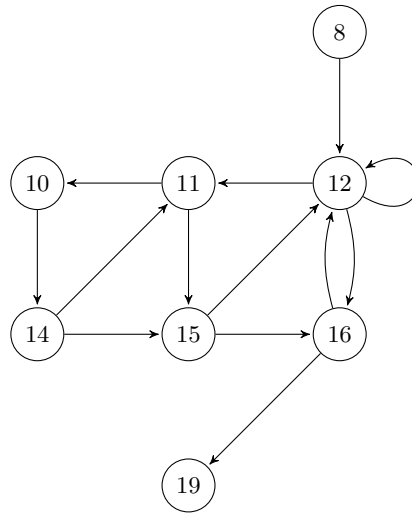


FIG. 3: A flow graph H whose stretching is shown in figure 4.

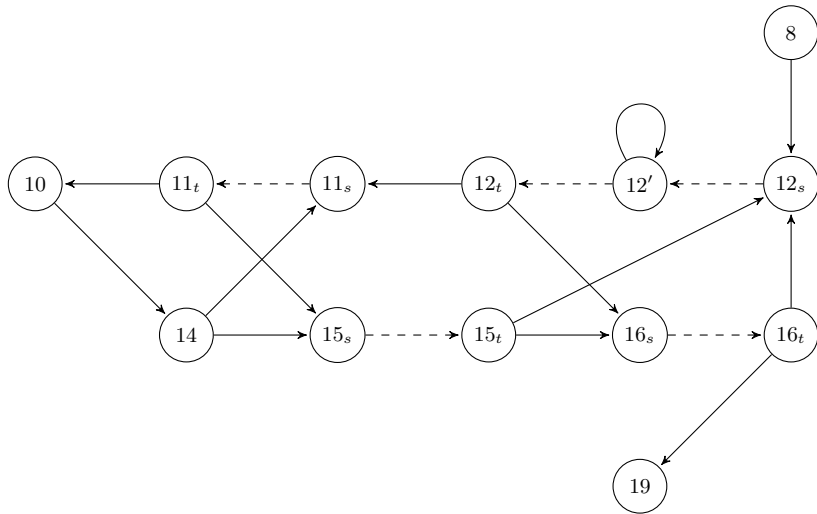


FIG. 4: The stretching of the flow graph H in figure 3. Edges introduced by the stretching are dashed.

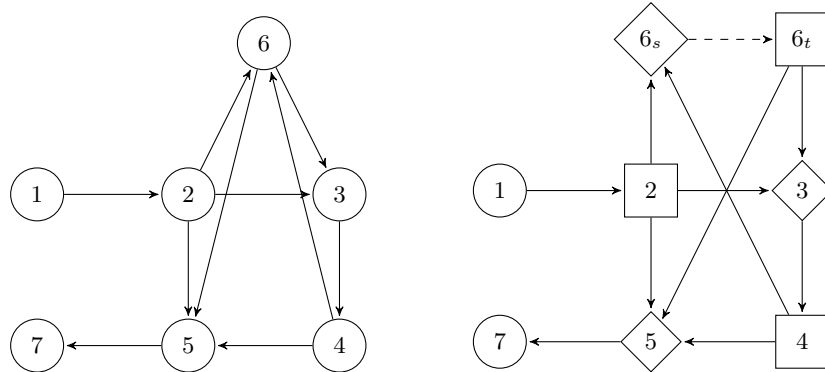


FIG. 5: A planar flow graph (L) whose stretching (R) is nonplanar. The bipartite structure of $K_{3,3}$ is indicated with squares and diamonds.

V. THE CATEGORY OF FLOW GRAPHS

The principal goal of this section is merely to motivate and justify the details of the sequel. The key points are the introduction of the category **Dgph** of digraphs, and of its full subcategory **Flow** whose objects are flow graphs. With this in mind, the casual reader may prefer to simply skip the remainder of this section.

It is natural to attempt to regard transformations of mathematical objects as morphisms in an appropriate category [12]. Unfortunately, in many if not most cases involving digraphs, such an attempt is complicated by technicalities that commonly arise from loops. [13] The basic problem is that while identifying vertices should induce a graph morphism, such a morphism should also preserve edges: in particular, it should preserve any edges between the vertices to be identified, necessarily inducing a loop.

The common way around this problem is to work with *reflexive digraphs* in which every vertex has a loop. However, flow graphs can contain loops at any subset of vertices except at the source and target, and so it is necessary in our context to treat the loops in a reflexive digraph as artifacts and incorporate “actual” loops in an auxiliary structure.

This reasoning motivates the following somewhat ugly definition: a *finite reflexive digraph with distinguished loops* or *L-graph* is given by an ordered triple (V, E, L) of vertices V , edges E , and loops L , where $L \subseteq V$, $|V| < \infty$, and $\Delta(V) \subseteq E \subseteq V \times V$, where as usual Δ denotes the diagonal functor. The *adjacency matrix* $A \equiv A(G)$ of the L-graph $G = (V, E, L)$ is given by

$$A_{jk} := \begin{cases} 1 & \text{if } (j, k) \in E \setminus \Delta(V) \text{ or } j = k \in L \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Each digraph with loops corresponds to a unique L-graph and vice versa: this bijective correspondence is specified by their common adjacency matrix. Henceforth we will at times use this correspondence implicitly and without further comment.

Recall that a morphism in the archetypal (di)graph category **Gph** is induced by $f : V \rightarrow V'$ via $f(j, k) := (f(j), f(k))$ for $(j, k) \in E$. In **Gph**, the set $\Delta(f(V)) \cap f(E)$ is precisely the set of pairs (j', j') in the image of E under f , and so in our context it is natural to consider something like $\pi(\Delta(f(V)) \cap f(E)) \cup f(L)$ as a candidate set of distinguished loops induced by a notional morphism that is in turn induced by f . (Here π merely denotes the canonical projection from a diagonal set.)

With this in mind, we will proceed to define a category **Lgph** whose objects are L-graphs. Morphisms between L-graphs are determined by maps between vertex sets as follows. Let $G = (V, E, L)$ be a L-graph. For $f : V \rightarrow V_f$, $(j, k) \in E \setminus \Delta(V)$, and $\ell \in L$, define (with a slight abuse of notation)

$$f(j, j) := (f(j), f(j)) \in E_f, \quad (6)$$

$$f(j, k) := \begin{cases} (f(j), f(k)) \in E_f & \text{if } f(j) \neq f(k) \\ f(j) \in L_f & \text{if } f(j) = f(k), \end{cases} \quad (7)$$

and $f(\ell) \in L_f$. Provided that $G_f := (V_f, E_f, L_f)$ is a L-graph, the augmented version of f defined just above will be an element of $\text{hom}_{\mathbf{Lgph}}(G, G_f)$, which will in turn consist only of maps of this form. Consequently, we may generally use the notation f to refer to either a map between L-graphs or their vertex sets.

LEMMA. **Lgph**, as provisionally defined just above, is indeed a category.

PROOF. In §B2. \square

COROLLARY. Let $G = (V, E, L)$ be a L-graph. For $(j, k) \in E \setminus \Delta(V)$, let $f|_{V \setminus \{k\}} = \text{id}_{V \setminus \{k\}}$ and $f(k) = j$, i.e., f identifies k with j . Write A and A' for the adjacency matrices of G and $(V, f(E), f(L))$, respectively. If $j \neq i$, $i' \neq k$ and $g, g' \in V$, then

$$\begin{aligned} A'_{ii'} &= A_{ii'}, \\ A'_{ij} &= A_{ij} \vee A_{ik}, \\ A'_{ji'} &= A_{ji} \vee A_{ki'}, \\ A'_{jj} &= A_{jj} \vee A_{jk} \vee A_{kj} \vee A_{kk}, \\ A'_{gk} &= 0, \\ A'_{kg'} &= 0. \end{aligned} \quad (8)$$

PROOF. The nontrivial action of f on edges is as follows: $(i, k) \mapsto (i, j)$, $(k, i) \mapsto (j, i)$, $(j, k) \mapsto j \in f(L)$, $(k, j) \mapsto j \in f(L)$, and $(k, k) \mapsto (j, j)$. Meanwhile, if $k \in L$ then $f(k) = j$. \square

This demonstrates that distinguished loops are not only preserved, but are formed whenever two neighboring vertices are identified. This is undesirable in (e.g.) the context of control flow of programs. However, it can be shown that the seemingly reasonable alternative behavior $A'_{jj} = A_{jj} \vee (A_{jk} \wedge A_{kj}) \vee A_{kk}$ would actually make vertex identification noncommutative, which would certainly be worse.

The solution is to form a category with the same objects as **Lgph** but more morphisms. Define the start and finish maps on vertices to be the identity function. The category **Dgph** has L-graphs as its objects and morphisms given by maps $f : (V, E, L) \rightarrow (V', E', L')$ which preserve starts and finishes (an equivalent but different construction of this category is in [13]). [37] In particular, edges can be mapped to vertices as well as edges and (distinguished) loops. The price we pay for this freedom is that morphisms are now only partially specified by their actions on vertices, and the following definition is essentially a convention about how to treat vertex identification without supplying any additional information.

We define **Flow** to be the full subcategory of **Dgph** whose objects are flow graphs in the obvious sense. [38]

VI. COARSENING FLOW GRAPHS

We begin this section with intuition: the coarsening $\odot G$ of a flow graph G is obtained by taking each leaf of its program structure tree and absorbing the interior of the corresponding sub-flow graph into its source. (See figure 6.) The details are below.

For $G = (V, E, L) \in \text{Ob}(\mathbf{Dgph})$, define the *absorption* of k into j to be the morphism in **Dgph** (or its image, depending on context) which corresponds to first applying the morphism in **Lgph** \subset **Dgph** obtained by identifying k with j (while retaining the original vertex set as in (8)) and then (if $k \neq j$) annihilating any loop at j (by mapping it to the vertex j). It is clear that first absorbing k and then m into j is equivalent to first absorbing m and then k into j . Consequently, for $U \subseteq V$ we may define the absorption of U into j in the obvious way. [39]

For $G, H \in \text{Ob}(\mathbf{Flow})$ with $H \subset G$ (the meaning of this should be evident), define the absorption of H to be the result of absorbing the interior of H into its source (considered as a vertex in G). This amounts to replacing H with a single edge between its source and target. Finally, define the *coarsening* $\odot G$ of G to be the result of absorbing all of the sub-flow graphs corresponding to leaves of the program structure tree of G . The fact that $\odot G$ is well-defined follows from [4] (cf. the “prime subprogram parse” of [14]) along with the preceding considerations. In particular, we have the following

LEMMA. Let $G \in \text{Ob}(\mathbf{Flow})$ and let $\odot G$ result from absorbing the vertex sets L_k into k for all $k \in K$ (so that L_k corresponds to a leaf of the program structure tree and $k \notin L_k$). Let $L := \cup_{k \in K} L_k$ (this set should not be confused with the set of loops in G) and $J := V \setminus (K \cup L)$, so that $V = J \cup K \cup L$ and J, K, L are mutually disjoint. Let $j, j' \in J$, $k, k' \in K$ with $k \neq k'$, and $\ell, \ell' \in L$. Finally, write $L_k^+ := \{k\} \cup L_k$ and let $g \in V$. Then the adjacency matrix of $\odot G$ w.r.t. the vertex set of G is A' , where

$$\begin{aligned} A'_{jj'} &= A_{jj'}, \\ A'_{jk'} &= \bigvee_{\ell' \in L_{k'}^+} A_{j\ell'}, \\ A'_{kj'} &= \bigvee_{\ell \in L_k^+} A_{\ell j'}, \\ A'_{kk'} &= \bigvee_{\ell \in L_k^+, \ell' \in L_{k'}^+} A_{\ell\ell'}, \\ A'_{kk} &= 0, \\ A'_{g\ell'} &= 0, \\ A'_{\ell g'} &= 0. \end{aligned} \tag{9}$$

PROOF. The result follows from the corollary in §V and the definitions of absorption and coarsening. \square

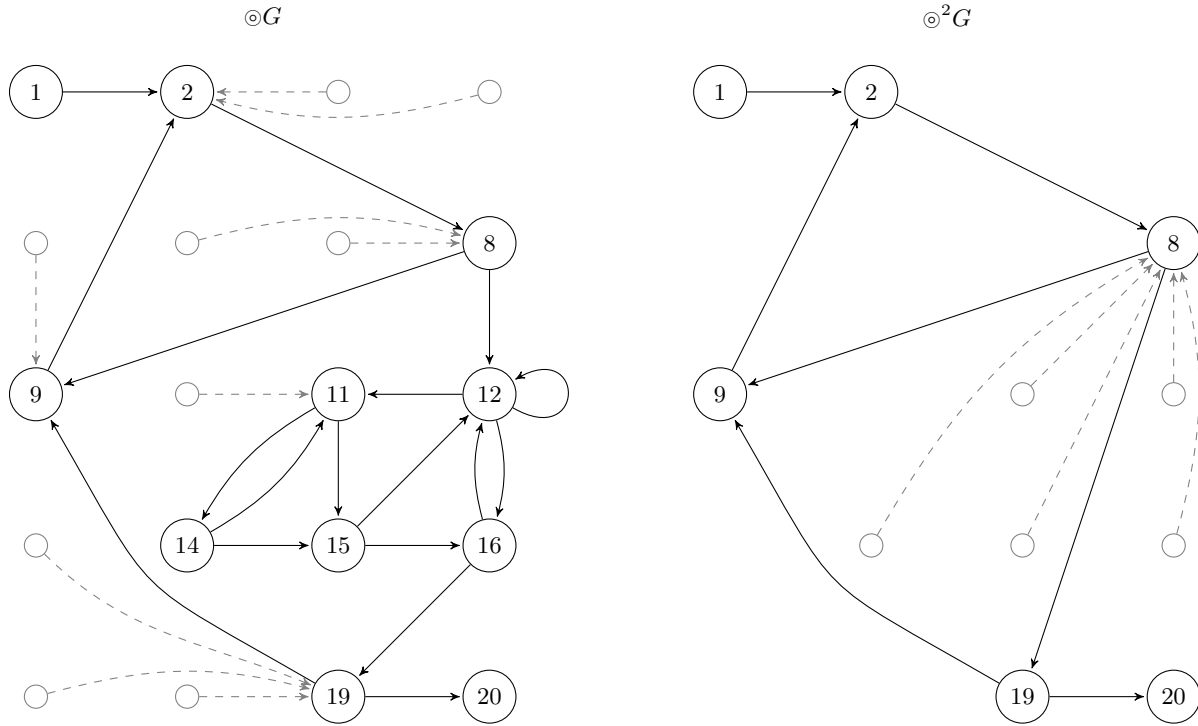


FIG. 6: Successive coarsenings of the flow graph G from figure 2. A third coarsening is trivial. (Note that the pullback of the diagram $a \xrightarrow{g \circ f} c \xleftarrow{g} b$ is $a \xleftarrow{id} a \xrightarrow{f} b$, so that f is the pullback of $g \circ f$ by g . We may therefore think of $\odot G$ somewhat literally as a kind of pullback of G by the leaves of its program structure tree.)

The real matter of substance in coarsening a flow graph is producing the sets J , K , and L referred to just above (it turns out to be easier to construct the L_k from L than to go in the opposite direction).

THEOREM. Using the notation of the preceding lemma, define a matrix M as follows. For each leaf (e_1, e_2) of the program structure tree, let $(e_1, e_2)^\circ$ denote its interior, and for all $j \in (e_1, e_2)^\circ$ set $M_{j, s(e_1)} = 1$. Then M is the adjacency matrix of a DAG (in fact, a forest) whose weakly connected components have vertex sets L_k^+ and corresponding targets k .

PROOF. Let (e_1, e_2) be a leaf of the PST. If $s(e_1)$ is in the interior of some other leaf (e'_1, e'_2) of the PST, then $e_1 = e'_2$. Therefore, $M_{s(e'_2), s(e'_1)} = 1$ and any other vertices j with $M_{j, s(e'_1)} = 1$ correspond to the remaining elements of $(e'_1, e'_2)^\circ$, which are leaves in the digraph G_M with adjacency matrix M . On the other hand, if $s(e_1)$ is not in the interior of some other leaf of the PST, then it is a target in G_M . The result follows. \square

Having considered coarsening flow graphs, we note that the appropriate mathematical formalization in the opposite direction—i.e., of inserting one flow graph into another [40]—is captured by the assertion that flow graphs form a (symmetric) *operad*. [15, 16] (Cf. [17, 18].) Let $P(n)$ denote the set of flow graphs with n ordered edges and define the following family of maps

$$\begin{aligned} \circ : P(n) \times P(k_1) \times \cdots \times P(k_n) &\rightarrow P(k_1 + \cdots + k_n) \\ (G, G_1, \dots, G_n) &\mapsto G \circ (G_1, \dots, G_n) \end{aligned} \quad (10)$$

by replacing, for each $1 \leq j \leq n$, the j th edge in G with G_j in the obvious way. For convenience, write $k_0 \equiv 0$. The edge ordering on $G \circ (G_1, \dots, G_n)$ is obtained by assigning edges $\sum_{i=0}^{j-1} k_i + 1, \dots, \sum_{i=0}^j k_i$ to $G_j \hookrightarrow G \circ (G_1, \dots, G_n)$ in the same order as the edges of G_j , i.e., the edge ordering is inherited from its local components.

THEOREM. The triple $\{e, \{P(n)\}_{n=1}^\infty, \circ\}$, where e denotes the flow graph with one edge, forms a operad (in **Set**).

PROOF. According to taste, either by straightforward definition-checking or direct analogy to other insertion operads, e.g. the little d -disks operad in **Top**. \square

Thus the operadic composition \circ and coarsening \odot operations are not only natural, but complementary:

PROPOSITION. If $G \in P(n)$ and $\odot G_j = e \neq G_j$ for $1 \leq j \leq n$, then $\odot(G \circ (G_1, \dots, G_n)) = G$. \square

VII. TENSORING FLOW GRAPHS

A. Tensoring in series

There is an essentially trivial tensor product on **Flow**. The idea is simply to identify, in the obvious way, the exit edge of the first flow graph with the entry edge of the second flow graph, i.e., to combine flow graphs in series. The reason that this tensor product structure is interesting and useful is that it allows us a way to model additional structure in an enriched category. Specifically, this leads to the **Flow**-category **SubFlow** $_G$ of sub-flow graphs of a flow graph G , and to a closely related strict ω -category.

We provide a quick sketch of the (very straightforward) details here. Let $f \in \text{hom}_{\mathbf{Flow}}(G, G_f)$ and $f' \in \text{hom}_{\mathbf{Flow}}(G', G'_{f'})$ with $V(G) \cap V(G') = \emptyset$. Define $G \boxtimes G'$ to be the flow graph obtained by identifying the exit edge of G and the entry edge of G' , and define $f \boxtimes f'$ to be the morphism in $\text{hom}_{\mathbf{Flow}}(G \boxtimes G', G_f \boxtimes G'_{f'})$ obtained by identifying the output of f on the exit edge of G with that of f' on the entry edge of G' .

PROPOSITION. **Flow** is a monoidal category with tensor product given by \boxtimes , and with unit object the flow graph e consisting of a single edge. \square

PROPOSITION. For a generic flow graph G , we can form a category **SubFlow** $_G$ enriched [19] over **Flow** as follows:

- $\text{Ob}(\mathbf{SubFlow}_G) := E(G)$; [41]
- for $e_s, e_t \in \text{Ob}(\mathbf{SubFlow}_G)$, the hom object $\mathbf{SubFlow}_G(e_s, e_t) \in \text{Ob}(\mathbf{Flow})$ is the (possibly empty) flow graph with entry edge e_s and exit edge e_t ;
- the composition morphism is induced by \boxtimes ;
- the identity element is determined by the flow graph e with one edge. \square

One important advantage in considering **SubFlow** $_G$ rather than the path category of G is that the former is finite (and the preceding sections essentially detail its construction), whereas the latter is infinite whenever there is a cycle in G . Another (related) advantage is that this perspective naturally yields a rare example of a strict ω -category [42], as we proceed to show. [43]

For $0 \leq m < \infty$, define

$$G(m) := \begin{cases} E(G) & \text{if } m = 0 \\ \{\mathbf{SubFlow}_G(e_1, e_2) : e_1, e_2 \in E(G)\} & \text{if } m > 0, \end{cases}$$

and define $\sigma_{1,0} : G(1) \rightarrow G(0)$ by $\sigma_{1,0}(\mathbf{SubFlow}_G(e_1, e_2)) := e_1$ and $\tau_{1,0} : G(1) \rightarrow G(0)$ by $\tau_{1,0}(\mathbf{SubFlow}_G(e_1, e_2)) := e_2$. For $2 \leq m < \infty$, define $\sigma_{m,m-1} : G(m) \rightarrow G(m-1)$ and $\tau_{m,m-1} : G(m) \rightarrow G(m-1)$ to be the identity maps. (When the domain and range are clear, we shall henceforth suppress the subscripts on $\sigma_{m,m-1}$ and $\tau_{m,m-1}$.) Then

$$G_{\rightrightarrows} := \dots G(m) \xrightleftharpoons[\tau]{\sigma} G(m-1) \xrightleftharpoons[\tau]{\sigma} \dots \xrightleftharpoons[\tau]{\sigma} G(1) \xrightleftharpoons[\tau]{\sigma} G(0)$$

is a ω -globular set [44].

We consider G_{\rightrightarrows} rather than a n -globular set for some finite n because $G(1)$ is equated with the set of sub-flow graphs of G , $G(2)$ with the set of sub-sub-flow graphs, etc.

Following [16], for $0 \leq p \leq m < \infty$, write $G(m) \times_{G(p)} G(m) := \{(g', g) \in G(m)^2 : \tau^{m-p}(g) = \sigma^{m-p}(g')\}$: then using Δ to denote the diagonal functor, $G(m) \times_{G(p)} G(m) = \Delta(G(m))$ unless $0 = p < m$, in which case $G(m) \times_{G(0)} G(m)$ is the set of pairs (g', g) of sub-flow graphs of G for which $g \boxtimes g'$ is also a sub-flow graph of G . For $0 \leq p < m$ define the *composition* $\bullet_p : G(m) \times_{G(p)} G(m) \rightarrow G(m)$ as follows:

$$\begin{aligned} g' \bullet_0 g &:= g \boxtimes g'; \\ p > 0 \Rightarrow g \bullet_p g &:= g. \end{aligned}$$

The identity $i_p : G(p) \rightarrow G(p+1)$ is just the inclusion map.

LEMMA. $(G \rightrightarrows, \bullet, i)$ is a strict ω -category.

PROOF. In §B3. \square

B. Tensoring in parallel

In this section we show that **Flow** carries a nontrivial monoidal structure (i.e., there is a tensor product operation that coherently combines flow graphs “in parallel” and not merely “in series” [20]). While the concept is rather obvious, the details are technical and consequently made explicit. In particular, although **Flow** is conceptually rather similar to the categories of n -cobordisms or tangles, the disjoint union only yields a tensor product in the latter cases: here, it must be modified to account for flow graphs whose entry and exit edges are identical or adjacent.

Let $s(e^+), t(e^+), s(e^-), t(e^-)$ be four fixed distinct points not contained in the vertex set of any graph already under consideration, so that $e^\pm := (s(e^\pm), t(e^\pm))$ may be regarded as two separated abstract edges. If G is a flow graph with entry edge e_s and (possibly adjacent or even identical) exit edge e_t , define a **Dgph**-morphism (i.e., the image may not be a flow graph) ϕ_G by the vertex/loop map

$$\phi_G(j) := \begin{cases} s(e^+) & \text{if } j = s(e_s) \\ t(e^+) & \text{if } j = t(e_s) \\ & \text{or } t(e_s) = s(e_t) \text{ and } j = t(e_t) \\ s(e^-) & \text{if } t(e_s) \neq s(e_t) \text{ and } e_s \neq e_t \text{ and } j = s(e_t) \\ t(e^-) & \text{if } t(e_s) \neq s(e_t) \text{ and } e_s \neq e_t \text{ and } j = t(e_t) \\ j & \text{otherwise} \end{cases} \quad (11)$$

along with the extension to edges determined by not sending edges to (diagonal/reflexive edges or) loops.

Intuitively, if the entry and exit edges of G are neither identical nor adjacent, then ϕ_G maps them respectively to e^+ and e^- : otherwise, ϕ_G maps the entry edge to e^+ and everything else (vertices and edges to the vertex; loops to a loop) to $t(e^+)$. The rationale for the latter case is that it is the only really generic and consistent way for us to complete the definition of such a nontrivial **Dgph**-morphism from a flow graph, and in fact this sort of definitional guidance is perhaps the primary rationale for invoking category theory *ab initio*.

PROPOSITION. With $j, k \in V(G)$ with $j \neq k$ and $j', k' \in V(G')$ with $j' \neq k'$, $\phi_G(j) = \phi_G(k) \Rightarrow \{j, k\} = \{t(e_s), t(e_t)\}$; similarly, $\phi_{G'}(j') = \phi_{G'}(k') \Rightarrow \{j', k'\} = \{t(e'_s), t(e'_t)\}$. \square

Now let $f \in \text{hom}_{\mathbf{Flow}}(G, G_f)$ and $f' \in \text{hom}_{\mathbf{Flow}}(G', G'_{f'})$ with $V(G) \cap V(G') = \emptyset$. We define

$$G \otimes G' := G \sqcup G' / \sim, \quad (12)$$

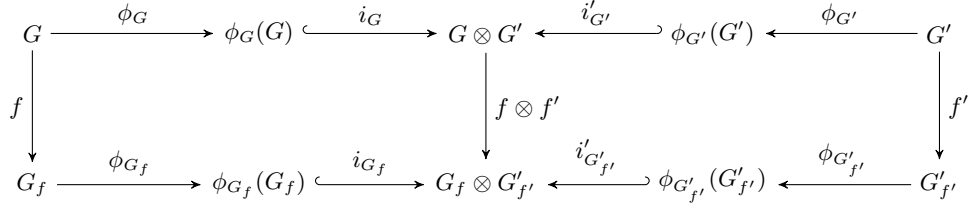
where the equivalence relation on the disjoint (graph) union is determined, for $j, k \in V(G)$ with $j \neq k$ and $j', k' \in V(G')$ with $j' \neq k'$, by

$$\begin{aligned} (j, 0) &\sim (j, 0) \text{ identically;} \\ (j', 1) &\sim (j', 1) \text{ identically;} \\ (j, 0) &\sim (k, 0) \iff (\phi_G(j) = \phi_G(k)) \wedge \star; \\ (j', 1) &\sim (k', 1) \iff (\phi_{G'}(j') = \phi_{G'}(k')) \wedge \star'; \\ (j, 0) &\sim (j', 1) \iff (\phi_G(j) = \phi_{G'}(j')) \wedge \star[j] \wedge \star[j']. \end{aligned} \quad (13)$$

where

$$\begin{aligned} \star &:= (t(e'_s) \neq s(e'_t)) \wedge (e'_s \neq e'_t); \\ \star' &:= (t(e_s) \neq s(e_t)) \wedge (e_s \neq e_t); \\ \star[j] &:= (t(e_s) = s(e_t)) \wedge (j \in \{t(e_s), t(e_t)\}) \Rightarrow \star; \\ \star[j'] &:= (t(e'_s) = s(e'_t)) \wedge (j' \in \{t(e'_s), t(e'_t)\}) \Rightarrow \star'. \end{aligned}$$

and with the obvious extension to edges. (Here e'_s and e'_t denote the entry and exit edges of G' .)

FIG. 7: The tensor product of morphisms in **Flow**.

LEMMA. (13) indeed defines an equivalence relation.

PROOF. In §B4. \square

PROPOSITION. If G and G' are flow graphs, then so is $G \otimes G'$. Furthermore, if e denotes the flow graph with a single edge, then $G \otimes e \cong e \otimes G \cong G$. \square

Thus in particular we have inclusions $i_G : \phi_G(G) \hookrightarrow G \otimes G'$ and $i'_{G'} : \phi_{G'}(G') \hookrightarrow G \otimes G'$ given respectively by $i_G(\phi_G(j)) = [(j, 0)]$ and $i'_{G'}(\phi_{G'}(j')) = [(j', 1)]$, where as per usual practice $[\cdot]$ indicates an equivalence class under \sim . $G \otimes G'$ is a flow graph formed by identifying the entry edges of G and G' , and identifying the respective exit edges if this does not affect the interiors of the factors, and otherwise collapsing the smaller factor in a way that sufficiently extends the identification of entry edges.

Meanwhile, we define $f \otimes f' \in \text{hom}_{\mathbf{Flow}}(G \otimes G', G_f \otimes G_{f'})$ as follows (see also figure 7):

$$(f \otimes f')(k) := \begin{cases} [(f(j), 0)] & \text{if } k = [(j, 0)] \\ [(f'(j'), 1)] & \text{if } k = [(j', 1)] \end{cases} \quad (14)$$

along with the implied extension to edges.

LEMMA. (14) is well-defined.

PROOF. We need to show that whenever $[(j, 0)] = [(j', 1)]$ we also have $[(f(j), 0)] = [(f'(j'), 1)]$. An equivalent assertion is that whenever $\phi_G(j) = \phi_{G'}(j')$, we also have $\phi_{G_f}(f(j)) = \phi_{G_{f'}}(f'(j'))$. There are precisely four cases in which the hypothesis can hold, corresponding to the first four cases of (11) (note that the second case has four subcases). In the first case, both $[(f(j), 0)]$ and $[(f'(j'), 1)]$ must be the source of the entry edge in $G_f \otimes G_{f'}$, since f and f' are morphisms in **Flow**; similarly, the other cases respectively give that both $[(f(j), 0)]$ and $[(f'(j'), 1)]$ must be the target of the entry edge, the source of the exit edge, and the target of the exit edge. \square

THEOREM. **Flow** is a monoidal category with tensor product given by (12) and (14), and with unit object the flow graph e consisting of a single edge.

PROOF. In §B5. \square

PROPOSITION. **Flow** is a symmetric monoidal category with \otimes as tensor product. \square

C. Remarks

The series and parallel tensor operations described above are very similar in spirit to the composition operations encountered in the study of so-called *series-parallel graphs* [21, 22], cf. [23]. While the categorical analysis of series and parallel tensor operations in the context of something like a system or wiring diagram has a very long history [24], a precise treatment appropriate to our development does not appear to be present in the literature.

Acknowledgments

The author thanks Brendan Fong, Artem Polyvyanyy, and David Spivak for their helpful comments.

Appendix A: Two-terminal graphs

Many of the considerations of the present paper have obvious analogues in the case of *two-terminal graphs* (TTGs). In particular, there is a multiresolution decomposition of TTGs described in [6, 14] (cf. [25]) that is a more granular version of the PST. This *refined process structure tree* is reducible via a straightforward graph transformation (“normalization”; a direct analogue of stretching) to the determination of the SPQR tree [7].

Unfortunately, computing SPQR trees is a notoriously intricate exercise: indeed, a correct linear time algorithm was not actually implemented until 2000 [26], though an incorrect version of the algorithm was described in 1973 [27]. Today there is still not a completely explicit description of the correct linear time algorithm in the literature: for such an account it is necessary to refer to one of the two known publically available implementations in the C++ OGDF and the Java jBPT frameworks. [45] [46] [47]

While the computation and properties of the fundamental decomposition for TTGs are more involved than the PST, some analogues of the constructions detailed in this paper are simpler since TTGs are defined to omit loops. On the other hand, one minor complication relative to flow graphs that informs notions of coarsening and inclusion operads for TTGs is that some TTGs can have their sources and targets swapped. A more significant (and perhaps surprising) complication is that it is not clear how to define a canonical parallel tensor operation for TTGs: the principal difficulty is the unit object. Lacking such an operation would be a significant shortcoming relative to the framework for flow graphs, as parallel tensoring in **Flow** corresponds to introducing an if/else statement in control flow, and in particular parallel tensoring with the flow graph $\rightarrow \rightleftarrows$ corresponds to introducing a do-while statement.

Appendix B: Technical proofs

1. Proof that the interiors of distinct canonical SESE regions are either disjoint or nested

[The proof is a straightforward adaptation of the original attempt from [4]; we also address a minor gap of the original attempt for case ii).]

Let (e_1, e_2) and (e'_1, e'_2) be distinct canonical SESE regions whose interiors are not disjoint, and let v be in their intersection. Since $e_1 \text{ dom } v$ and $e'_1 \text{ dom } v$, it must be that either $e_1 \text{ dom } e'_1$ or $e'_1 \text{ dom } e_1$: assume the former w.l.o.g. Similarly, since $e_2 \text{ pdom } v$ and $e'_2 \text{ pdom } v$, either $e_2 \text{ pdom } e'_2$ or $e'_2 \text{ pdom } e_2$: in the former case, $(e'_1, e'_2) \subset (e_1, e_2)$ and we are done, so assume the latter case. We now have three cases to consider: i) $e_2 = e'_1$; ii) $e_2 \neq e'_1$ and $e'_1 \text{ dom } e_2$; and iii) $e_2 \neq e'_1$ and e'_1 does not dominate e_2 . We shall show that each of these three cases leads to a contradiction.

Case i). Since in this case $e_2 = e'_1$, we have that $e_2 \text{ dom } v$ and $e_2 \text{ pdom } v$, so it must be that any path from the source to the target that traverses v must contain a cycle of the form $(e_2, \dots, v, \dots, e_2)$. But this means that v cannot be in the interior of (e_1, e_2) , a contradiction: hence case i) cannot hold.

Case ii). Since in this case $e'_1 \text{ dom } e_2$ and generically $e_1 \text{ dom } e'_1$, any path γ_{02} from the source to e_2 may be decomposed (using an obvious notation) as $\gamma_{02} \equiv \gamma_{01}\gamma_{11'}\gamma_{1'2}$. Meanwhile since $e_2 \text{ pdom } e_1$, any path $\gamma_{1\infty}$ from e_1 to the target can be decomposed as $\gamma_{1\infty} \equiv \gamma_{12}\gamma_{2\infty}$. Taken together, these decompositions imply that any path from the source to the target that traverses e'_1 can be decomposed as $\gamma_{01}\gamma_{11'}\gamma_{1'2}\gamma_{2\infty}$, so that $e_2 \text{ pdom } e'_1$ and $e'_1 \text{ pdom } e_1$.

Moreover, if there is a cycle that traverses e_1 , it also traverses e_2 and *vice versa*, so we may write such a cycle as $\omega_{12} \equiv \gamma_{12}\gamma_{21}$, where $\gamma_{12} \equiv \gamma_{11'}\gamma_{1'2}$ as above. Hence such a cycle ω_{12} must traverse e'_1 . Similarly, if there is a cycle that traverses e'_1 , it also traverses e'_2 and *vice versa*, so we may write such a cycle as $\omega_{1'2'} \equiv \gamma_{1'2'}\gamma_{2'1'}$, where $\gamma_{1'2'}$ traverses e_2 since $e'_2 \text{ pdom } e_2$. Hence such a cycle $\omega_{1'2'}$ must traverse e_2 . It follows that (e'_1, e_2) is a SESE region.

Since both (e_1, e_2) and (e'_1, e'_2) are canonical SESE regions, we have that $e_1 \text{ pdom } e'_1$ and $e'_2 \text{ dom } e_2$. At the same time, $e'_1 \text{ pdom } e_1$, so it must be that $e_1 = e'_1$. It follows that (e_1, e'_2) is also a SESE region, and therefore also that $e_2 \text{ dom } e'_2$, so it must be that $e_2 = e'_2$. This contradicts the hypothesis that (e_1, e_2) and (e'_1, e'_2) are distinct: hence case ii) cannot hold.

Case iii). Since in this case e'_1 does not dominate e_2 , there is a path γ_{02} from the source to e_2 that avoids e'_1 . Suppose that e'_1 does not postdominate e_2 , i.e., suppose that there is a path $\gamma_{2\infty}$ from e_2 to the target that avoids e'_1 . Then since $e'_2 \text{ pdom } e_2$, $\gamma_{2\infty}$ must traverse e'_2 . But since $e'_1 \text{ dom } e'_2$ and the concatenated path $\gamma \equiv \gamma_{02}\gamma_{2\infty}$ from the source to the target traverses e'_2 , it must be that $\gamma_{2\infty}$ traverses e'_1 , contradicting the assumption that e'_1 does not postdominate e_2 . Therefore since $e'_1 \text{ pdom } e_2$ and $e_2 \text{ pdom } v$, we have that $e'_1 \text{ pdom } v$. Moreover, $e'_1 \text{ dom } v$, so it

must be that any path from the source to the target that traverses v must contain a cycle of the form $(e'_1, \dots, v, \dots, e'_1)$. But this means that v cannot be in the interior of (e'_1, e'_2) , a contradiction: hence case iii) cannot hold. \square

2. Proof that **Lgph** is a category

This is a simple exercise in bookkeeping. There are only three things to check: the existence of identity morphisms, composability of morphisms, and the associativity of composition. The first of these three is trivial. The remaining two are nearly so: indeed, only the behavior for edges needs to be checked, since the behavior for vertices and loops is just inherited from composition in **Set**.

Now consider $G \xrightarrow{f} G_f \xrightarrow{g} G_g \xrightarrow{h} G_h$, $(j, k) \in E$, and the following cases:

- i. $j = k$,
- ii. $j \neq k$ and $f(j) = f(k)$,
- iii. $f(j) \neq f(k)$ and $g(f(j)) = g(f(k))$,
- iv. $g(f(j)) \neq g(f(k))$ and $h(g(f(j))) = h(g(f(k)))$,
- v. $h(g(f(j))) \neq h(g(f(k)))$.

To check composability of morphisms, we merely need to show that the composition (in **Set**) of the morphisms corresponding to the vertex maps f and g (is defined whenever the vertex map $g \circ f$ is and) equals the morphism (again, in **Set**) corresponding to the vertex map $g \circ f$. This easy but tedious exercise is reproduced here. The composition of the morphisms corresponding to the vertex maps f and g behaves as follows for the cases above:

- i. $(j, j) \xrightarrow{f} (f(j), f(j)) \in E_f \xrightarrow{g} (g(f(j)), g(f(j))) \in E_g$,
- ii. $(j, k) \xrightarrow{f} f(j) \in L_f \xrightarrow{g} g(f(j)) \in L_g$,
- iii. $(j, k) \xrightarrow{f} (f(j), f(k)) \in E_f \xrightarrow{g} g(f(j)) \in L_g$,
- iv. [same as case v.] $(j, k) \xrightarrow{f} (f(j), f(k)) \in E_f \xrightarrow{g} (g(f(j)), g(f(k))) \in E_g$.

Meanwhile, the morphism corresponding to the vertex map $g \circ f$ respectively behaves as

- i. $(j, j) \xrightarrow{g \circ f} (g(f(j)), g(f(j))) \in E_g$,
- ii. $(j, k) \xrightarrow{g \circ f} g(f(j)) \in L_g$,
- iii. $(j, k) \xrightarrow{g \circ f} g(f(j)) \in L_g$,
- iv. [same as case v.] $(j, k) \xrightarrow{g \circ f} (g(f(j)), g(f(k))) \in E_g$.

These behaviors are equivalent to their counterparts, and the composability of morphisms follows.

Checking the associativity of composition is similar. For $h \circ (g \circ f)$, we have

- i. $(j, j) \xrightarrow{g \circ f} (g(f(j)), g(f(j))) \in E_g \xrightarrow{h} (h(g(f(j))), h(g(f(j)))) \in E_h$,
- ii. [same as case iii.] $(j, k) \xrightarrow{g \circ f} g(f(j)) \in L_g \xrightarrow{h} h(g(f(j))) \in L_h$,
- iv. $(j, k) \xrightarrow{g \circ f} (g(f(j)), g(f(k))) \in E_g \xrightarrow{h} h(g(f(j))) \in L_h$.
- v. $(j, k) \xrightarrow{g \circ f} (g(f(j)), g(f(k))) \in E_g \xrightarrow{h} (h(g(f(j))), h(g(f(k)))) \in E_h$.

Meanwhile for $(h \circ g) \circ f$, we have

- i. $(j, j) \xrightarrow{f} (f(j), f(j)) \in E_f \xrightarrow{h \circ g} (h(g(f(j))), h(g(f(j)))) \in E_h$,
- ii. $(j, k) \xrightarrow{f} f(j) \in L_f \xrightarrow{h \circ g} h(g(f(j))) \in L_h$,

- iii. [same as case iv.] $(j, k) \xrightarrow{f} (f(j), f(k)) \in E_f \xrightarrow{h \circ g} h(g(f(j))) \in L_h$.
- v. $(j, k) \xrightarrow{f} (f(j), f(k)) \in E_f \xrightarrow{h \circ g} (h(g(f(j))), h(g(f(k)))) \in E_h$.

As before, these behaviors are equivalent to their counterparts, and the associativity of composition follows. \square

3. Proof that $(G_{\Rightarrow}, \bullet, i)$ is a strict ω -category

We check the six axioms in the same manner as they are presented in [16]. For $g \in G(p)$, we write $1_g := i_p(g)$.

The first thing to check is the sources and targets of composites: for $p > 0$ this is trivial, so we only need to consider the two cases $(p, m) = (0, 1)$ and $(p, m) = (0, 2)$. In the first case, $\sigma(g' \bullet_0 g) = \sigma(g \boxtimes g') = \sigma(g)$ as required; τ behaves similarly. In the second case, $\sigma(g' \bullet_0 g) = \sigma(g \boxtimes g') = g \boxtimes g'$ and $\sigma(g') \bullet_0 \sigma(g) = \sigma(g) \boxtimes \sigma(g') = g \boxtimes g'$, so that $\sigma(g' \bullet_0 g) = \sigma(g') \bullet_0 \sigma(g)$ as required; again, τ behaves similarly.

The second thing to check is the sources and targets of identities. If $p > 0$ this follows from $\sigma(1_g) = g = \tau(1_g)$; otherwise, this follows from $\sigma(1_e) = \sigma(e) = e = \tau(e) = \tau(1_e)$ for an edge e .

The third thing to check is associativity. If $p > 0$ this is trivial; otherwise, this follows from the associativity of \boxtimes .

The fourth thing to check is identities. If $p > 0$ this is trivial; otherwise, this follows from $g \bullet_0 \sigma(g) = \sigma(g) \boxtimes g = g \boxtimes \tau(g) = \tau(g) \bullet_0 g$.

The fifth thing to check is binary interchange. If $0 < q < p$ this is trivial; otherwise, this follows from $(g' \bullet_0 g) \bullet_p (g' \bullet_0 g) = g' \bullet_0 g$.

The sixth and final thing to check is nullary interchange. If $0 < q < p$ this follows from $1_g \bullet_q 1_g = 1_g = 1_{g \bullet_q g}$; otherwise, this follows from $1_{g'} \bullet_0 1_g = g \boxtimes g' = 1_{g' \bullet_0 g}$. \square

4. Proof that (13) defines an equivalence relation

Since it is obvious from the structure of $\star[j]$ and $\star[j']$ that $(j', 1) \sim (j, 0) \iff (j, 0) \sim (j', 1)$, the only thing to show is transitivity. A (perhaps unnecessarily) brutally straightforward proof consists of verifying each of the eight assertions

$$(\ell_{1,b_1}, b_1) \sim (\ell_{2,b_2}, b_2) \sim (\ell_{3,b_3}, b_3) \Rightarrow (\ell_{1,b_1}, b_1) \sim (\ell_{3,b_3}, b_3)$$

for $(b_1, b_2, b_3) \in \{0, 1\}^3$ and $\ell_{1,b_1}, \ell_{2,b_2}, \ell_{3,b_3}$ distinct.

First, consider $(b_1, b_2, b_3) = (0, 0, 0)$: we must show in this case that

$$(\phi_G(\ell_{10}) = \phi_G(\ell_{20}) = \phi_G(\ell_{30})) \wedge \star \Rightarrow (\phi_G(\ell_{10}) = \phi_G(\ell_{30})) \wedge \star,$$

but this is trivial.

Next, consider $(b_1, b_2, b_3) = (0, 0, 1)$. Here we must show that

$$(\phi_G(\ell_{10}) = \phi_G(\ell_{20}) = \phi_{G'}(\ell_{31})) \wedge \star \wedge \star[\ell_{20}] \wedge \star[\ell_{31}] \Rightarrow (\phi_G(\ell_{10}) = \phi_{G'}(\ell_{31})) \wedge \star[\ell_{10}] \wedge \star[\ell_{31}],$$

By the proposition above, $\{\ell_{10}, \ell_{20}\} = \{t(e_s), t(e_t)\}$, so $t(e_s) = s(e_t)$ and $\star[\ell_{10}]$ is true, establishing the desired result.

For $(b_1, b_2, b_3) = (0, 1, 0)$, the desired implication is

$$(\phi_G(\ell_{10}) = \phi_{G'}(\ell_{21}) = \phi_G(\ell_{30})) \wedge \star[\ell_{10}] \wedge \star[\ell_{21}] \wedge \star[\ell_{30}] \Rightarrow (\phi_G(\ell_{10}) = \phi_G(\ell_{30})) \wedge \star.$$

By the proposition above, $\{\ell_{10}, \ell_{30}\} = \{t(e_s), t(e_t)\}$, so $t(e_s) = s(e_t)$ and $\ell_{10}, \ell_{30} \in \{t(e_s), t(e_t)\}$. Since in the present case both $\star[\ell_{10}]$ and $\star[\ell_{30}]$ are true by assumption and we have just shown their hypotheses true, their mutual conclusion \star is also true here. This yields the desired implication. (NB. Although $\star[\ell_{21}]$ is true in this case, neither its hypothesis nor its conclusion are.)

By symmetry, the last case we need to consider is $(b_1, b_2, b_3) = (0, 1, 1)$: we need to show here that

$$(\phi_G(\ell_{10}) = \phi_{G'}(\ell_{21}) = \phi_{G'}(\ell_{31})) \wedge \star[\ell_{10}] \wedge \star[\ell_{21}] \wedge \star' \Rightarrow (\phi_G(\ell_{10}) = \phi_{G'}(\ell_{31})) \wedge \star[\ell_{10}] \wedge \star[\ell_{31}].$$

By the proposition, $\{\ell_{21}, \ell_{31}\} = \{t(e'_s), t(e'_t)\}$, so $t(e'_s) = s(e'_t)$ and $\star[\ell_{31}]$ is true, so we are done. \square

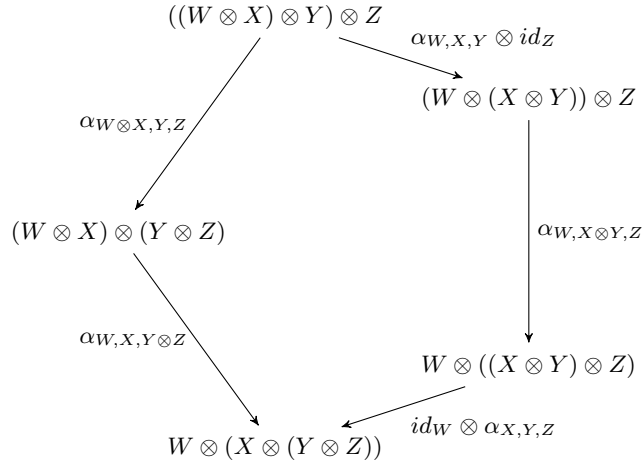


FIG. 8: The pentagon equation.

5. Proof that Flow is a monoidal category with tensor product given by (12) and (14), and with unit object the flow graph e consisting of a single edge

We must establish two things: that \otimes is a bifunctor, and that it satisfies the necessary coherence conditions. To see that \otimes is a bifunctor, first note that

$$(id_G \otimes id_{G'})([(j, 0)]) = [(j, 0)] = id_{G \otimes G'}([(j, 0)])$$

by (14), and similarly

$$(id_G \otimes id_{G'})([(j', 1)]) = [(j', 1)] = id_{G \otimes G'}([(j', 1)]),$$

so that $id_G \otimes id_{G'} = id_{G \otimes G'}$. Now we must show that $(g \otimes g') \circ (f \otimes f') = (g \circ f) \otimes (g' \circ f')$. But this is easily seen since, again by (14), we have

$$(g \otimes g')([(f(j), 0)]) = [(g(f(j)), 0)] = ((g \circ f) \otimes (g' \circ f'))([(j, 0)])$$

and

$$(g \otimes g')([(f'(j'), 1)]) = [(g'(f'(j')), 1)] = ((g \circ f) \otimes (g' \circ f'))([(j', 1)]).$$

So (since the action on edges follows trivially) \otimes is in fact a bifunctor.

To see that the putative tensor product is coherent, we first note that the triangle equation turns out to be trivial, so we need only verify the pentagon equation, which is recalled in figure 8. The associator $\alpha_{G, G', G''} : (G \otimes G') \otimes G'' \rightarrow G \otimes (G' \otimes G'')$ is given by

$$\alpha_{G, G', G''} : \begin{cases} [([(j, 0)], 0)] & \mapsto [(j, 0)] \\ [([(j', 1)], 0)] & \mapsto [([(j', 0)], 1)] \\ [(j'', 1)] & \mapsto [([(j'', 1)], 1)] \end{cases} \quad (\text{B1})$$

along with the implied extension to edges. The explicit form of (B1) makes it clear that the associator is bijective, and hence an isomorphism.

For notational convenience, let W, X, Y, Z denote flow graphs with $(w, x, y, z) \in V(W) \times V(X) \times V(Y) \times V(Z)$. Now

$$\begin{array}{ccccc} [([(w, 0)], 0)] & & [([(w, 0)], 0)] & & [(w, 0)] \\ [([(x, 1)], 0)] & \xleftarrow{\alpha_{W \otimes X, Y, Z}} & [([(x, 1)], 0)] & \xleftarrow{\alpha_{W, X, Y \otimes Z}} & [([(x, 0)], 1)] \\ [([(y, 1)], 0)] & & [([(y, 0)], 1)] & & [([(y, 0)], 1)] \\ [(z, 1)] & & [([(z, 1)], 1)] & & [([(z, 1)], 1)] \end{array} \quad (\text{B2})$$

Meanwhile,

$$\begin{array}{c}
 [([([w, 0]), 0]), 0)] \\
 [([([x, 1]), 0]), 0)] \\
 [([([y, 1]), 0]), 0)] \\
 [(z, 1)]
 \end{array}
 \xrightarrow{\alpha_{W,X,Y} \otimes id_Z}
 \begin{array}{c}
 [([w, 0]), 0)] \\
 [([([x, 0]), 1]), 0)] \\
 [([([y, 1]), 1]), 0)] \\
 [(z, 1)]
 \end{array}
 \xrightarrow{\alpha_{W,X \otimes Y,Z}}
 \begin{array}{c}
 [(w, 0)] \\
 [([([x, 0]), 0]), 1)] \\
 [([([y, 1]), 0]), 1)] \\
 [([([z, 1]), 1)])
 \end{array}
 \xrightarrow{id_W \otimes \alpha_{X,Y,Z}}
 \begin{array}{c}
 [(w, 0)] \\
 [([([x, 0]), 1)]) \\
 [([([y, 0]), 1]), 1)] \\
 [([([z, 1]), 1]), 1)]
 \end{array} \quad (B3)$$

The pentagon equation follows from the equality of the rightmost parts of (B2) and (B3), as does the theorem. \square

Appendix C: Examples

The examples in this section use code from §D. To run our code (and these examples), MatlabBGL is required, though we provide native MATLAB code for computing the dominance relation in §E that can facilitate workarounds.

1. Program structure tree example

The information in figures 1 and 2 may be reproduced via the following command sequence:

```
% Form an adjacency matrix
G = sparse(20,20);
edges = [1,2;5,2;2,3;3,4;9,5;7,6;7,7;8,7;4,8;6,9;13,9;11,10;12,11;14,11;...
        8,12;12,12;15,12;16,12;17,13;10,14;11,15;14,15;12,16;15,16;18,17;...
        19,18;16,19;19,20];
for j = 1:size(edges,1)
    G(edges(j,1),edges(j,2)) = 1;
end

% Compute the program structure tree
temp = pst(G);
```

To wit, `temp.bdry` returns the edges enclosing SESE regions:

1	2	19	20
2	3	3	4
2	3	4	8
3	4	4	8
9	5	5	2
7	6	6	9
8	7	6	9
8	7	7	6
11	10	10	14
8	12	16	19
17	13	13	9
18	17	13	9
18	17	17	13
19	18	13	9
19	18	17	13
19	18	18	17

Meanwhile, `temp.msr` returns the vertices enclosing minimal SESE regions:

1	20
2	19
3	3
4	4
5	5
6	6
7	7

```

10    10
12    16
13    13
17    17
18    18

```

Finally, `temp.pst` returns the program structure tree:

```

1      2
2      3
2      4
2      5
2      6
2      7
2     12
2     13
2     17
2     18
12    10

```

2. Stretching example

Continuing from §C 1, the information in figures 3 and 4 may be reproduced as follows:

```

ind = [8,10:12,14:16,19]; H = G(ind,ind);
S = stretchflowgraph(H);

```

whereupon `ind(S.ofgv)` yields

```

8    10    11    11    12    12    12    14    15    15    16    16    19

```

which corresponds to the ordered tuple $(8, 10, 11_s, 11_t, 12_s, 12'_t, 12_t, 14, 15_s, 15_t, 16_s, 16_t, 19)$. Meanwhile, `S.A` yields

```

(4,2)    1
(7,3)    1
(8,3)    1
(3,4)    1
(1,5)    1
(10,5)   1
(12,5)   1
(5,6)    1
(6,6)    1
(6,7)    1
(2,8)    1
(4,9)    1
(8,9)    1
(9,10)   1
(7,11)   1
(10,11)  1
(11,12)  1
(12,13)  1

```

Thus, for instance, the edge $(10, 5)$ in `S.A` corresponds to $(15_t, 12_s)$ in the stretching of H , and to $(15, 12)$ in H itself.

To reproduce the results of figure 5, use

```

P = sparse(7,7);
edges = [1,2;2,3;2,5;2,6;3,4;4,5;4,6;5,7;6,3;6,5];
for j = 1:size(edges,1)
    P(edges(j,1),edges(j,2)) = 1;
end
test_planar_graph(max(P,P')) % returns 1 (note symmetrization of argument)
SP = stretchflowgraph(P);
test_planar_graph(max(SP.A,SP.A')) % returns 0 (note symmetrization of argument)

```

3. Coarsening example

The left portion of figure 6 can be reproduced via `C = coarsening(G)`, which yields

```
A: [20x20 double]
J: [1 12 14 15 16 20]
K: [2 8 9 11 19]
L: [3 4 5 6 7 10 13 17 18]
M: [20x20 double]
N: [20x20 double]
```

The adjacency matrix of the coarsening is `C.A`:

```
(1,2)      1
(9,2)      1
(2,8)      1
(8,9)      1
(19,9)     1
(12,11)    1
(14,11)    1
(8,12)     1
(12,12)    1
(15,12)    1
(16,12)    1
(11,14)    1
(11,15)    1
(14,15)    1
(12,16)    1
(15,16)    1
(16,19)    1
(19,20)    1
```

and the “absorption matrix” `C.M` is

```
(3,2)      1
(4,3)      1
(6,7)      1
(7,8)      1
(5,9)      1
(10,11)    1
(13,17)    1
(17,18)    1
(18,19)    1
```

The right portion of the figure, corresponding to a second coarsening, requires us to re-index: viz.

```
ind = union(C.J,C.K) % equals union(find(sum(C.A,1)),find(sum(C.A,2)))
```

yields

```
1      2      8      9      11      12      14      15      16      19      20
```

whereupon

```
CA = C.A(ind,ind);
CC = coarsening(CA)
```

gives

```
A: [11x11 double]
J: [1 2 4 10 11]
K: 3
L: [5 6 7 8 9]
M: [11x11 double]
N: [11x11 double]
```

with `[I1,I2] = find(CC.A); [ind(I1)',ind(I2)']` producing

```

1      2
9      2
2      8
8      9
19     9
8      19
19     20

```

Appendix D: MATLAB code

1. Dominator matrix of a flow graph

```

function y = dominatormatrix(A,s)

% Dominator matrix of flow graph A with start node s.
% Produces a sparse matrix D w/ D(j,k) = 1 iff j dominates k.
% Uses MatlabBGL lengauer_tarjan_dominator_tree function.

n = size(A,1);
if size(A,2) ~= n
    error('A must be square');
end
if numel(setdiff(unique(A),[0,1])) || nnz(A) == 0
    error('A must be a nontrivial 0-1 matrix');
end

%% Dominator tree using MatlabBGL
% NB. The call "lengauer_tarjan_dominator_tree(sparse(0),1)" would
% typically cause MATLAB to crash as of April 2014; this is however
% prevented by a check above ensuring that nnz(A) ~= 0.
tree = lengauer_tarjan_dominator_tree(A,s);

%% Reasonably elegant backtrack
D = sparse(n,n);
k = find(diag(D)==0,1); % k = 1 is the number of the first zero column
while numel(k)
    %% Fill in the kth column of D by backtracking through the tree
    D(k,k) = 1;
    idom = tree(k);
    while idom ~= 0
        D(idom,k) = 1;
        idom = tree(idom);
    end
    %% Induced column fill-in corresponding to visited nodes
    newcol = D(:,k);
    newcol(k) = 0;
    k2 = tree(k);
    while any(newcol)
        D(:,k2) = newcol;
        newcol(k2) = 0;
        k2 = tree(k2);
    end
    %% Update k
    k = find(diag(D)==0,1);
end

%% Redundant/slow backtrack shown here for comparison only
% D2 = sparse(n,n);
% for k = 1:n
%     dom{k} = k;
%     idom = tree(k);
%     while idom ~= 0
%         dom{k} = union(dom{k},idom);
%         idom = tree(idom);
%     end

```

```
%      D2(dom{k},k) = 1;      % dom{d} = find(D2(:,d))';
% end

%% Output
y = D;
```

2. Standardized encapsulation of a generic adjacency matrix

```
function y = encapsulation(A)

% This code produces a struct containing the (adjacency matrix of an)
% encapsulation of (the adjacency matrix corresponding to) a digraph in the
% field A, and an index array pointing to vertices of the original digraph
% in the field i.
%
% Given a generic digraph G with at least one source and at least one
% target, there is a unique minimal digraph containing G with a single
% (entry) edge from a unique source and a single (exit) edge to a unique
% target. Call this digraph the encapsulation of G.
%
% The first and last vertices of the output correspond to the source and
% target, respectively. The encapsulation is formed by inserting single
% source and target vertices and edges if necessary. In the event that the
% original underlying digraph has no sources (resp. targets), a source will
% be forced to the first original vertex (resp. last original vertex), and
% a warning will be displayed.
%
% More generally, warnings will also be displayed, and remedial action
% taken, if the input is not already of the form encapsulation(B) for some
% digraph B, i.e., if B is not a flow graph with first vertex the source
% and last vertex the target.
%
% Note that if y = encapsulation(A) and fyi = find(y.i), then
% y.A(fyi,fyi) equals A(y.i(fyi),y.i(fyi))

%% Preliminaries and basic error handling
A = sparse(A);
n = size(A,1);
index = 1:n; % keeps track of original vertices
if size(A,2) ~= n
    error('A must be square');
end
if numel(setdiff(unique(A),[0,1]))
    warning(['A should be a 0-1 matrix: producing a substitute with ',...
            'the same sparsity pattern']);
    A = A./(A+(A==0));
end

%% Vertex degree characteristics
% NB. d0, dp, and dm are all columns
d0 = diag(A); % number of loops
A0 = A-diag(d0);
dp = sum(A0,1)'; % number of incoming edges excluding any loop
dm = sum(A0,2); % number of outgoing edges excluding any loop

%% Source
sources = find(dp==0); % loops don't materially bear on sourceness
if numel(sources) == 0
    % Prepend a new source vertex and add a single edge from it to the
    % vertex previously having index 1
    A = [sparse(n+1,1),[1,sparse(1,n-1);A]];
    index = [0,index];
    warning('no original source vertices');
elseif numel(sources) == 1
    % If the source has precisely one non-loop outbound edge and no
    % loop, then...
    if dm(sources) == 1 && d0(sources) == 0
        if sources ~= 1
```

```

        % ...move the source vertex to the first position...
        perm = [sources,1:(sources-1),(sources+1):n];
        A = A(perm,perm);
        index = index(perm);
        warning('original source vertex not at index 1');
    end
else
    % ...otherwise, prepend a new source vertex and add a single edge
    % to the original source
    A = [sparse(n+1,1),[(dp==0)';A]];
    index = [0,index];
    warning('no original single entry edge');
end
else
    % Prepend a new source vertex and add edges from it to each original
    % source vertex...
    A = [sparse(n+1,1),[(dp==0)';A]];
    index = [0,index];
    % ...then prepend yet another source vertex and add a single edge from
    % it to the source vertex previously introduced just above
    A = [sparse(n+2,1),[1,sparse(1,n);A]];
    index = [0,index];
    warning('more than one original source vertex');
end

%% Encore of vertex degree characteristics for convenience
% NB. d0, dp, and dm are all columns
d0 = diag(A); % number of loops
A0 = A-diag(d0);
dp = sum(A0,1)'; % number of incoming edges excluding any loop
dm = sum(A0,2); % number of outgoing edges excluding any loop
n = size(A,1);

%% Target
targets = find(dm==0); % loops don't materially bear on targetness
if numel(targets) == 0
    % Append a new target vertex and add a single edge to it from the
    % vertex (just) previously having index n
    A = [A,[sparse(n-1,1);1];sparse(1,n+1)];
    index = [index,0];
    warning('no original target vertices');
elseif numel(targets) == 1
    % If the target has precisely one non-loop inbound edge and no
    % loop, then...
    if dp(targets) == 1 && d0(targets) == 0
        if targets ~= n
            % ...move the target vertex to the last position...
            perm = [1:(targets-1),(targets+1):n,targets];
            A = A(perm,perm);
            index = index(perm);
            warning('original target vertex not at last index');
        end
    else
        % ...otherwise, append a new target vertex and add a single edge
        % from the original target
        A = [[A,(dm==0)];sparse(1,n+1)];
        index = [index,0];
        warning('no original single exit edge');
    end
else
    % Append a new target vertex and add edges to it from each original
    % target vertex...
    A = [[A,(dm==0)];sparse(1,n+1)];
    index = [index,0];
    % ...then append yet another target vertex and add a single edge to it
    % from the target vertex previously introduced just above
    A = [[A,[sparse(n,1);1];sparse(1,n+2)]];
    index = [index,0];
    warning('more than one original target vertex');
end
end

```

```

%% Output
y.A = A;
y.i = index;

```

3. Stretching a flow graph

```

function y = stretchflowgraph(A)

% Produces the stretching of a flow graph. More specifically, given a flow
% graph A, this returns a struct y containing two fields:
% * y.ofgv, an array each of whose entries corresponds to a vertex of y.A
%   and points in turn to the corresponding original flow graph vertex
% * y.A, the adjacency matrix of the stretching of A

%% Safety/encapsulation checks
E = encapsulation(A);
fEi = find(E.i);
if any(size(E.A)-size(A))
    error('A is not a flow graph');
else
    if any(E.A(fEi,fEi)-A(E.i(fEi),E.i(fEi)))
        error('A is not a flow graph');
    end
end

%% Vertex degree characteristics
% NB. d0, dp, and dm are all columns
d0 = diag(A); % number of loops
A0 = A-diag(d0);
dp = sum(A0,1)'; % number of incoming edges excluding any loop
dm = sum(A0,2); % number of outgoing edges excluding any loop

%% Vertex degree tests
% NB. tp, tm, and t0 are all columns since dp, dm, and d0 are also...
tp = (dp>1);
tm = (dm>1);
t0 = (d0==1);
% ...which makes this cleaner:
t = [tp,tm,t0]; % test array

%% Vertex cases for the stretching lemma
% We have the following pairs (hash(v),slc), where slc0 is the stretching
% lemma case corresponding to the value of hash(v): (3,1), (5,2), (6,3),
% and (7,4)
hash = t*[4;2;1];
slc0 = zeros(size(hash));
slc0(hash==3) = 1;
slc0(hash==5) = 2;
slc0(hash==6) = 3;
slc0(hash==7) = 4;
slc0 = slc0'; % make it a row

%% Loop initializations
% N is the number of vertices during (and after) the stretching procedure
N = numel(slc0);
% ofgv0 is an array of pointers to original flow graph vertices
ofgv0 = 1:N;
% Dynamically growing (one might even say "stretching"[!]) copy of A
S0 = A;

%% Main loop
while any(slc0)
    v = find(slc0,1);
    u = 1:(v-1);
    w = (v+1):N;
    S1 = sparse(N,N);
    %% Furnish new vertex partition {U,V,W} and populate S1(V,V)

```

```

if slc0(v) == 1
    V = [v,v+1];    % V = [vs,vt]
    U = u;
    W = w+1;
    % loop @ V(1); edge from V(1) to V(2)
    S1(V,V) = [1,1;0,0];
elseif slc0(v) == 2
    V = [v,v+1];    % V = [vs,vt]
    U = u;
    W = w+1;
    % loop @ V(2); edge from V(1) to V(2)
    S1(V,V) = [0,1;0,1];
elseif slc0(v) == 3
    V = [v,v+1];    % V = [vs,vt]
    U = u;
    W = w+1;
    % Edge from V(1) to V(2)
    S1(V,V) = [0,1;0,0];
elseif slc0(v) == 4
    V = [v,v+1,v+2];    % V = [vs,vp,vt]
    U = u;
    W = w+2;
    % loop @ V(2); edges from V(1) to V(2) and V(2) to V(3)
    S1(V,V) = [0,1,0;0,1,1;0,0,0];
end
%% Finish the update from S0 to S1 and back
% First handle vertices that are unaffected other than indexing
S1(U,U) = S0(u,u);
S1(U,W) = S0(u,w);
S1(W,U) = S0(w,u);
S1(W,W) = S0(w,w);
% Next handle incoming non-loop edges
S1(U,V(1)) = S0(u,v);
S1(W,V(1)) = S0(w,v);
% Next handle outgoing non-loop edges
S1(V(end),U) = S0(v,u);
S1(V(end),W) = S0(v,w);
% Back to S0
S0 = S1;
%% Update from ofgv0 to ofgv1 and back
ofgv1(U) = ofgv0(u);
ofgv1(V) = ofgv0(v);
ofgv1(W) = ofgv0(w);
ofgv0 = ofgv1;
%% Update from slc0 to slc1 and back
slc1(U) = slc0(u);
slc1(V) = 0;    % now no stretching lemma case applies on V
slc1(W) = slc0(w);
slc0 = slc1;
%% Update N
N = numel(slc0);
end

%% Output
y.ofgv = ofgv0;
y.A = S0;

```

4. Identification of SESE edge boundaries in a flow graph

```

function y = sesebdry(A)

% Identification of single-entry/single-exit (SESE) regions in a flow
% graph. The first two columns of the output give the sources and targets
% of edges entering SESE regions; the second two columns give the sources
% and targets of edges leaving the corresponding regions.
% The approach here is based on the slow algorithm in Johnson, Pearson, and
% Pingali (the fast algorithm does not appear to be particularly well
% suited to MATLAB).

```

```

% Uses MatlabBGL dfs and lengauer_tarjan_dominator_tree functions
% explicitly and implicitly, respectively.

%% Preliminaries
% If A is not already a flow graph with first and last vertices the entry
% and exit, respectively, then enforce that
A = getfield(encapsulation(A),'A');

% Add an edge from the exit to the entry
S = A;
S(size(S,1),1) = 1;

% Form an undirected multigraph by making edges undirected. Note that for
% any two vertices, at most two multigraph edges connect them, though it is
% possible for both of these edges to be back edges w.r.t. a DFS tree. Note
% also that the diag(diag(S)) term is necessary to avoid introducing
% spurious multiple loops
U = S+S'-diag(diag(S)); % NOT max(S,S')--this wouldn't produce a multigraph
n = size(U,1); % number of vertices

%% Form dominator matrix for later use
D = dominatormatrix(A,1); % D(j,k) = 1 iff j dominates k

%% Perform a DFS on U
% MatlabBGL function dfs returns depth (unassigned), discovery/finish times
% (dt/ft), and predecessors (pred)
[~,dt,ft,pred] = dfs(U,1);

%% DFS tree
dfstree = sparse(n,n);
for t = 2:n
    dfstree(pred(t),t) = 1;
end
% Undirected DFS tree
udfstree = max(dfstree,dfstree');

%% Undirected graph of back edges
% Note that it is possible for both edges in a double edge to be back
% edges. In this case they can't correspond to a SESE region--i.e., two
% back edges between the same vertices will NOT be cycle equivalent. For
% this reason it is imperative to account for edge multiplicity when
% checking for cycle equivalence via brackets below.
back = U-udfstree;

%% Form arrays for undirected tree and back edges
% [T1,T2] is an array of undirected tree edges s.t. T2 is sorted
T1 = pred(2:n)'; % first tree edge vertex
T2 = (2:n)'; % second tree edge vertex
% Array of undirected back edges, WITHOUT repetition by multiplicity
[i1,i2] = find(tril(back)); % convenient to use tril so that i2 is sorted
% Array of undirected back edges, WITH repetition by multiplicity
B1 = []; % first back edge vertex
B2 = []; % second back edge vertex
for a1 = 1:numel(i1) % loop over vertex pairs
    for a2 = 1:back(i1(a1),i2(a1)) % loop over edge multiplicity
        B1 = [B1;i1(a1)];
        B2 = [B2;i2(a1)];
    end
end
nT = n-1; % number of tree edges: first edge is a phantom edge from "0"
nB = numel(B1); % number of back edges
N = nT+nB; % N = numel(V1) = numel(V2)

%% Create an array [V1,V2] whose rows are edges of S in a convenient order
% The first nT rows are the representatives in S of dfstree, in order; the
% last nB rows are the representatives in S of back edges
V1 = [T1;B1]; % first tree/back edge vertex (not for an S-edge yet)
V2 = [T2;B2]; % second tree/back edge vertex (not for an S-edge yet)
% Orient rows of [V1,V2] as edges of S by iterative comparison/deletion
S0 = S;

```

```

for j = 1:N
    if S0(V1(j),V2(j)) % do nothing
    elseif S0(V2(j),V1(j)) % swap
        temp = V1(j);
        V1(j) = V2(j);
        V2(j) = temp;
    else % error
        error('[V1,V2] swap error');
    end
    S0(V1(j),V2(j)) = S0(V1(j),V2(j))-1;
end

%% Find brackets of each DFS tree edge t
for t = 2:n % tth tree edge is (pred(t),t); the first edge is a phantom
    %% Ancestors of pred(t), including pred(t) itself
    % Note that the number of paths in dfstree from j to k is at most 1 and
    % is given by the (j,k)th entry of I/(I-T)-I, where I = speye(n) and
    % T is the matrix incarnation of the DFS tree given by
    % T = sparse(n,n); for t = 2:n, T(pred(t),t) = 1; end
    % This gives an easy alternative mechanism for computing ancestors and
    % descendants below, and might even be useful in the event that
    % MatlabBGL's dfs function is unavailable and a simpler DFS (and,
    % necessarily in the event, a dominator routine) is implemented from
    % scratch.
    ancestors = [];
    current = pred(t);
    while current >= 1
        ancestors = [current,ancestors];
        current = pred(current);
    end
    %% Descendants of t, including t itself
    % Use parenthesis theorem for DFS, which implies that the descendants
    % of t are given by all u s.t. dt(t) <= dt(u) and ft(u) <= ft(t)
    descendants = intersect(find(dt>=dt(t)),find(ft<=ft(t)))';
    %% Bitmasks (column vectors)
    maskA = zeros(n,1);
    maskA(ancestors) = 1;
    maskD = zeros(n,1);
    maskD(descendants) = 1;
    %% Brackets are backedges from descendants to ancestors
    % First form some projections
    projA = spdiags(maskA,0,n,n);
    projD = spdiags(maskD,0,n,n);
    % Use projections to isolate the brackets. It is very convenient to
    % represent brackets in sparse matrices of fixed size for comparison
    % later. The first term on the RHS below produces back edge
    % representatives (i.e., edges in S, not U) from ancestors to
    % descendants; the second term produces back edge representatives from
    % descendants to ancestors. Note that the entrywise products with S
    % are in effect a sparsity mask ensuring we get only the
    % representatives (vs. their oppositely directed edges).
    bracket{t} = (projA*back*projD).*S + (projA*back'*projD)'.*S;
end

%% Compare brackets of tree edges to determine their cycle equivalence in U
CE = sparse(N,N); % cycle equivalence matrix initialization
% Use theorem 5 of the JPP paper
for t1 = 1:(nT-1)
    % Use a +1 to account for nT = n-1 / the phantom edge
    bra1 = bracket{t1+1};
    for t2 = (t1+1):nT
        bra2 = bracket{t2+1};
        if nnz(bra1-bra2) == 0
            CE(t1,t2) = 1; % t1th and t2th tree edges are c.e.
        end
    end
end
end

%% Compare tree and back edges to determine their cycle equivalence in U
% Use theorem 4 of the JPP paper

```

```

BEs = nT+(1:nB);    % set of back edges
for t = 1:nT
    % Use a +1 to account for nT = n-1 / the phantom edge
    bra = bracket{t+1};
    if nnz(bra) == 1
        [e1,e2] = find(bra);
        if bra(e1,e2) == 1 % then this is the sole bracket, so it is c.e.
            % Try both possibilities to find the edge index
            % NB. The outer intersection is necessary to ensure b is a back
            % edge and not a tree edge
            b = intersect(intersect(find(V1==e1),find(V2==e2)),BEs);
            if numel(b) ~= 1
                error('b should have one element');
            end
            CE(t,b) = 1;    % tth tree edge and bth back edge are c.e.
        end
    end
end

%% Map cycle equivalences in U to edge enclosures of SESE regions in A
% By theorem 2 of the JPP paper, c.e. in U equates to edge enclosure of a
% SESE region in A. If edges E1 = [j1,k1] and E2 = [j2,k2] are c.e. in U,
% then either E1 dominates E2 (so that k1 dominates j2 dominates k2) or
% vice versa (so that k2 dominates j1 dominates k1). In the first case, say
% that the SESE region is enclosed by edges E1 and E2/nodes k1 and j2; in
% the second case, by E2 and E1/k2 and j1.
% The intent of this code cell is to produce an array with four columns
% s.t. the first two columns are the source and target of the incoming edge
% and the second two columns are the source and target of the outgoing edge
% of a SESE region. There is no attempt to make the row ordering useful.
[ce1,ce2] = find(CE);
nCE = numel(ce1);
bdry = zeros(nCE,4);
for e = 1:nCE
    j1 = V1(ce1(e));
    k1 = V2(ce1(e));
    j2 = V1(ce2(e));
    k2 = V2(ce2(e));
    if D(j1,j2)
        if k2 ~= 1 % this is to avoid the edge from n to 1
            bdry(e,:) = [j1,k1,j2,k2];
        end
    elseif D(j2,j1)
        if k1 ~= 1 % this is to avoid the edge from n to 1
            bdry(e,:) = [j2,k2,j1,k1];
        end
    else
        error('dominator analysis failed!');
    end
end
% Eliminate zero rows due to avoiding the edge from n to 1
bdry = bdry(any(bdry,2),:);

%% Output
% Note that the rows are sorted by enclosing vertex pairs
[~,i] = sortrows(bdry(:,[2,3]));
y = bdry(i,:);

```

5. Program structure tree of a flow graph

```

function y = pst(A)

% Given a flow graph A with entry node 1 and exit node size(A,1), returns
% the edge boundaries of SESE regions (y.bdry); vertex boundaries of
% minimal SESE regions (y.MSR), and program structure tree (y.pst)

%% Preliminaries
% If A is not already a flow graph with first and last vertices the entry

```

```

% and exit, respectively, then enforce that
A = getfield(encapsulation(A),'A');
n = size(A,1);

%% Perform a DFS on A
% MatlabBGL/GAIMC function dfs returns depth (unassigned), discovery/finish
% times (dt/unassigned), and predecessors (pred)
[~,dt,~,pred] = dfs(A,1);

%% Identify edges enclosing SESE regions
% The first two columns of bdry give the sources and targets of edges
% entering SESE regions; the second two columns give the sources and
% targets of edges leaving the corresponding regions
bdry = sesebdry(A);

%% Identify minimal SESE regions (MSRs) using DFS discovery rank
M = sparse(n,n);
M(1,n) = 1; % this is a minimal SESE region by definition
[~,i] = sort(dt);
[~,dr] = sort(i); % dr(j) is DFS discovery rank of vertex j; dr(i(j)) = j
for j = 1:n
    % Find target vertices of SESE regions with source dr(j)
    SRs = bdry(bdry(:,2)==dr(j),3);
    if numel(SRs)
        % The minimal SESE region with source dr(j) has target with least
        % discovery rank
        target = SRs(dr(SRs)==min(dr(SRs)));
        M(dr(j),target) = 1;
    end
end
[SM,TM] = find(M);
% The first column of msr gives the targets of edges entering MSRs; the
% second column gives the sources of edges exiting the same MSRs
msr = sortrows([SM,TM]);

%% Build the PST by traversing DFS tree
parent = zeros(1,n);
current = zeros(1,n);
current(1) = 1;
for j = 2:n
    s = pred(i(j));
    t = i(j);
    % Check for entry into a SR
    entering = any((bdry(:,1)==s).*(bdry(:,2)==t));
    % Check for exit from a SR
    exiting = any((bdry(:,3)==s).*(bdry(:,4)==t));
    % Assign parent and current SRs
    if entering && exiting
        parent(t) = parent(s);
        current(t) = t;
    elseif entering
        parent(t) = current(s);
        current(t) = t;
    elseif exiting
        % parent(1) := 0 and also parent(parent(1)) := 0
        parent(t) = parent(max(parent(s),1));
        current(t) = parent(s);
    else
        parent(t) = parent(s);
        current(t) = current(s);
    end
end
% Form the adjacency matrix P of the PST
P = sparse(n,n);
for j = 2:n-1
    if parent(j)
        P(parent(j),current(j)) = 1;
    end
end
% Extract the (parent,child) edges of P

```

```

[PSTP,PSTC] = find(P);
% The first column of pst gives the parent "header"; the second column
% gives the corresponding child "header"
pst = sortrows([PSTP,PSTC]);

%% Output
y.bdry = bdry;
y.msr = msr;
y.pst = pst;

```

6. Interior of a SESE region

```

function y = seseinterior(A,seser)

% Get vertices of the interior of the SESE region of A corresponding to
% seser, which should be an array of the form [se1,te1,se2,te2], where if
% (e1,e2) are the edges defining the region, their sources and targets are
% respectively indicated. I.e., if p = pst(A), then seser should be a row
% of p.bdry.
%
% NB. Earlier attempts to do this using dominator matrices foundered,
% probably due to a subtle error in definition 6 of [Johnson, Pearson, and
% Pingali]: cf. [Boissinot, Brisk, Darté, and Rastello]. Note however that
% this error does not substantively affect the remainder of the PST
% framework when the definition of interior implicit here is used.

Atemp = A;
% Cut the SESE region's entry and exit edges
Atemp(seser(1),seser(2)) = 0;
Atemp(seser(3),seser(4)) = 0;
% Now do a DFS of the cutout:
[d,~,~,~] = dfs(Atemp,seser(2));
% The interior is just where we went along with where we started from
interior = union(seser(2),find(d>0)');

y = interior;

```

7. Coarsening a flow graph

```

function y = coarsening(A)

% Produces the coarsening of a flow graph. Besides the adjacency matrix
% (A), the vertex partition into unaffected (J), absorbing (K), and
% absorbed (L) vertices is returned, along with the adjacency matrix (M) of
% a forest encoding the absorption process and a supplementary
% matrix (N) giving the number of paths between vertices in the
% forest.
%
% A is assumed to be a flow graph with entry node 1 and exit node size(A,1)

n = size(A,1);

%% Program structure tree
temp = pst(A);
BDRY = temp.bdry;
MSR = temp.msr;
PST = temp.pst;

%% Get leaves L0 of the PST
ind1 = PST(:,1);
ind2 = PST(:,2);
T = sparse(n,n);
for j = 1:numel(ind1)
    T(ind1(j),ind2(j)) = 1;
end
LO = intersect(ind2,find(sum(T,2)==0)');

```

```

%% Construct the adjacency matrix for the "absorption DAG"
M = sparse(n,n);
for ell = 1:numel(L0)
    % First, explicitly find the minimal SESE region (there can be only
    % one) corresponding to L0(ell)
    te1 = L0(ell);
    se2 = MSR(MSR(:,1)==te1,2);
    % Second, explicitly find the incoming and outgoing edges
    seser = BDRY((BDRY(:,2)==te1)&(BDRY(:,3)==se2),:);
    % Third, find the interior of the SESE region
    interior = seseinterior(A,seser);
    % M is the adjacency matrix of a DAG, each of whose weakly connected
    % components has a single sink that is an absorbing vertex; the other
    % vertices in a given WCC are absorbed into that sink. Furthermore,
    % the structure of M reflects the partial order governing the
    % permissible absorptions of individual vertices.
    M(interior,seser(1)) = 1;
end

%% Find the sets of absorbed (L), absorbing (K), and other (J) vertices
% K is the set of sinks for M. Note that a sink must have an inbound edge
K = find((sum(M,2)==0)'*(sum(M,1)));
L = find(sum(M,2))';
J = setdiff(1:n,union(K,L));

%% Break out absorbees by absorbers
N = speye(n)/(speye(n)-M); % N(i1,i2) = # of paths in M from i1 to i2
NO = N-diag(diag(N)); % omit paths of length zero (there are no cycles)
for k = 1:numel(K)
    Lk{K(k)} = find(NO(:,K(k)));
end

%% Coarse adjacency matrix
A2 = A;
A2(J,J) = A(J,J);
for k = 1:numel(K)
    Lkp = union(K(k),Lk{K(k)});
    A2(J,K(k)) = max(A(J,Lkp), [], 2);
    A2(K(k),J) = max(A(Lkp,J), [], 1);
    for k2 = 1:numel(K)
        if k2 ~= k
            Lk2p = union(K(k2),Lk{K(k2)});
            A2(K(k),K(k2)) = max(max(A(Lkp,Lk2p)));
        end
    end
    A2(K(k),K(k)) = 0; % superfluous and gratuitous
end
A2(:,L) = 0;
A2(L,:) = 0;

%% Output
y.A = A2;
y.J = J;
y.K = K;
y.L = L;
y.M = M;
y.N = N;

```

Appendix E: Supplement: native MATLAB code to compute dominance relations

1. Specialized dataflow dominator code

Note that this requires some plumbing to form a dominator matrix, cf. §D 1.

```

function y = dominators(A,s)

% Dominators--a la the dragon book.

```

```

% * A is the adjacency matrix of a flow graph.
% * s is the specified start node.
% Cf. dominatorsgeneric.m for a version more readily adaptable to generic
% data flow problems.
%
% FOR A FASTER WAY TO COMPUTE DOMINATORS, USE THE FUNCTION
% lengauer_tarjan_dominator_tree in MatlabBGL

A = sparse(A);
n = size(A,1);
if size(A,2) ~= n
    error('A must be square');
end
if numel(setdiff(unique(A),[0,1])) || nnz(A) == 0
    error('A must be a nontrivial 0-1 matrix');
end

%% Initialization (and boundary)
oldout = cell(1,n);
out = cell(1,n);
changes = zeros(1,n);
preds = cell(1,n);
for B = 1:n
    oldout{B} = 1:n;
    if B == s
        %% Boundary
        out{B} = s;
    else
        out{B} = 1:n;
    end
    changes(B) = numel(setxor(out{B},oldout{B}));
    preds{B} = find(A(:,B));
end

%% Main loop
in = cell(1,n);
while any(changes)
    for B = 1:n
        oldout{B} = out{B};
        if B == s
            % do nothing
        else
            in{B} = out{preds{B}(1)};
            for P = 2:numel(preds{B})
                in{B} = intersect(in{B},out{preds{B}(P)});
            end
            %% Transfer function
            out{B} = union(in{B},B);
        end
        changes(B) = numel(setxor(out{B},oldout{B}));
    end
end

y = out;

```

2. Generic dataflow dominator code

Note that this requires some plumbing to form a dominator matrix, cf. §D 1.

```

function y = dominatorsgeneric(A,node,dir,bdry,init)

% Dominators using a fairly generic iterative data-flow algorithm as per
% the purple dragon book (fig. 9.40), therefore requiring several auxiliary
% arguments:
% * A is the adjacency matrix of a flow graph (a generic digraph should be
%   modified to be a flow graph if necessary).
% * node is respectively an entry or exit, depending on whether dir is
%   nonzero (forward) or zero (backward).
% * bdry is the "boundary", e.g. node

```

```

% * init is the initialization, e.g. 1:N (generally, the top element w.r.t.
%   the lattice meet operation defined as a local function below)
% Note that the lattice meet operation and transfer functions are defined
% as local functions for ease of adaptation to other data-flow problems.
% This is also the reason for the high abstraction level of the code as
% compared to the implementation in dominators.m.
%
% Example:
% A1 = sparse(10); A1(1,[2,3]) = 1; A1(2,3) = 1; A1(3,4) = 1;
% A1(4,[3,5,6]) = 1; A1(5,7) = 1; A1(6,7) = 1; A1(7,[4,8]) = 1;
% A1(8,[3,9,10]) = 1; A1(9,1) = 1; A1(10,7) = 1;
% node = 1; dir = 1; bdry = node; init = 1:size(A1,1);
% temp = dominatorsgeneric(A1,node,dir,bdry,init);
% dom = temp.z;

%% Preliminaries
A = sparse(A);
n = size(A,1);
if size(A,2) ~= n
    error('A must be square');
end
if numel(setdiff(unique(A),[0,1])) || nnz(A) == 0
    error('A must be a nontrivial 0-1 matrix');
end

%% Initialization (and boundary)
% B indicates a basic block (node); a and z respectively indicate in/out
% sets in a way consistent with the direction dir
oldz = cell(1,n);
z = cell(1,n);
changes = zeros(1,n);
opadj = cell(1,n);
for B = 1:n
    oldz{B} = init;
    if B == node
        z{B} = bdry;
    else
        z{B} = init;
    end
    changes(B) = numel(setxor(z{B},oldz{B}));
    %% Adjacent nodes in opposite direction
    if dir
        opadj{B} = find(A(:,B)); % predecessors
    else
        opadj{B} = find(A(B,:)); % successors
    end
end

%% Main loop
a = cell(1,n);
while any(changes)
    for B = 1:n
        oldz{B} = z{B};
        if B == node
            % do nothing
        else
            a{B} = z{opadj{B}(1)};
            for C = 2:numel(opadj{B})
                a{B} = lop(a{B},z{opadj{B}(C)});
            end
            z{B} = trans(a{B},B);
        end
        changes(B) = numel(setxor(z{B},oldz{B}));
    end
end

%% Output
y.a = a; % = "in" if dir is nonzero, otherwise = "out"
y.z = z; % = "out" if dir is nonzero, otherwise = "in"

```



```

end

%% LOCAL FUNCTIONS

%% Lattice meet operation
function y = lop(x1,x2)

y = intersect(x1,x2);

end

%% Transfer function
function y = trans(x,B)

y = union(x,B);

end

```

-
- [1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2010.
 - [2] K. D. Cooper and L. Torczon, *Engineering a Compiler, 2nd ed.* Morgan Kaufmann, 2012.
 - [3] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
 - [4] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: computing control regions in linear time," in *PLDI*, 1994.
 - [5] B. Boissinot, P. Brisk, A. Darte, and F. Rastello, "SSI properties revisited," *ACM TECS*, vol. 11S, no. 21, 2012.
 - [6] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," *Data Knowl. Eng.*, vol. 68, p. 793, 2009.
 - [7] A. Polyvyanyy, J. Vanhatalo, and H. Völzer, "Simplified computation and generalization of the refined process structure tree," *Lect. Notes. Comp. Sci.*, vol. 6551, p. 25, 2011.
 - [8] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Prog. Lang. Sys.*, vol. 1, p. 115, 1979.
 - [9] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Softw. Pract. Exper.*, vol. 4, p. 1, 2001.
 - [10] L. Georgiadis, R. E. Tarjan, and R. F. Werneck, "Finding dominators in practice," *J. Graph Algorithms App.*, vol. 10, p. 69, 2006.
 - [11] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan, "Finding dominators via disjoint set union," *J. Discrete Algorithms*, vol. 23, p. 2, 2013.
 - [12] S. Mac Lane, *Categories for the Working Mathematician, 2nd ed.* Springer, 2010.
 - [13] R. Brown, I. Morris, J. Shrimpton, and C. D. Wensley, "Graphs of morphisms of graphs," *Electronic J. Combinatorics*, vol. 15, no. A1, 2008.
 - [14] R. E. Tarjan and J. Valdes, "Prime subprogram parsing of a program," in *POPL*, 1980.
 - [15] M. Markl, S. Shnider, and J. Stasheff, *Operads in Algebra, Topology and Physics*. AMS, 2002.
 - [16] T. Leinster, *Higher Operads, Higher categories*. Cambridge, 2004.
 - [17] D. I. Spivak, "The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits," 2013.
 - [18] D. Rupel and D. I. Spivak, "The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes," 2013.
 - [19] G. M. Kelly, *Basic Concepts of Enriched Category Theory*. Cambridge, 1982.
 - [20] B. Coecke, Ed., *New Structures for Physics*. Springer, 2010.
 - [21] R. J. Duffin, "Topology of series-parallel networks," *J. Math. Anal. Appl.*, vol. 10, p. 303, 1965.
 - [22] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer, 2009.
 - [23] D. J. Dougherty and C. Gutiérrez, "Normal forms for binary relations," *Theor. Comp. Sci.*, vol. 360, p. 228, 2006.
 - [24] E. S. Bainbridge, "Feedback and generalized logic," *Info. Control*, vol. 31, p. 75, 1976.
 - [25] S. Fugishige, "Canonical decompositions of symmetric submodular systems," *Disc. Appl. Math.*, vol. 5, p. 175, 1983.
 - [26] C. Gutwenger and P. Mutzel, "A linear-time implementation of SPQR-trees," *Lect. Notes. Comp. Sci.*, vol. 1984, p. 77, 2001.
 - [27] J. Hopcroft and R. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Comp.*, vol. 2, p. 135, 1973.
 - [28] Y. I. Manin, "Renormalization and computation I: motivation and background," 2009.
 - [29] C. Delaney and M. Marcolli, "Dyson-Schwinger equations in the theory of computation," 2013.
 - [30] P. Gaucher, "A model category for the homotopy theory of concurrency," *Homology, Homotopy App.*, vol. 5, p. 549, 2003.
 - [31] Y. H. Tsin, "Decomposing a multigraph into split components," in *Proc. 18th. Ann. Australasian Th. Symp.* Australian Comp. Soc., 2012.
 - [32] To wit, assume that A is a sparse adjacency matrix representing a DAG with a single source at the first index.

Then the following works: `n = size(A,1); I = speye(n); D = sparse(n,n); N = I/(I-A); for k = 1:n, for j = 1:n, if N(1,k) == N(1,j)*N(j,k), D(j,k) = 1; end, end, end`

- [33] NB. One sometimes sees variants of the definition and naming of this particular sort of concept, for the latter most typically as “flowgraph”, “flowchart”, or “flow chart”. Some concepts with the same name are technically quite different but “spiritually” viewed in a similar context, as, e.g., in the work of Manin. [28, 29]
- [34] NB. Degenerate SESE regions of the form (e_1, e_1) are excluded by the original definition of [4]. We avoid this exclusion so as to make the series tensor product of §VII A work nicely.
- [35] A useful restatement of this is that if (e_1, e_2) is a SESE region with $e_2 \neq e_3$ and $e_2 \text{ dom } e_3$, then (e_1, e_3) is not a SESE region unless (e_2, e_3) is a SESE region.
- [36] Cf. the critically different definition 6 of [4] and our code in §D 6, wherein the interior of a SESE region is (implicitly) defined as the set of vertices reachable from $t(e_s)$ after deleting e_s and e_t .
- [37] **Dgph** is similar but not identical to the category of reflexive multigraphs: the principal difference is that objects of the former do not have multiple edges/loops.
- [38] As pointed out by D. Spivak, it would be desirable to describe flow graphs in terms of **Lgph** or **Dgph**, e.g. as algebras for some monad. Unfortunately, the author’s primitive understanding of category theory is presently inadequate to this task.
- [39] Failing to make fixed choices about whether to preserve or annihilate loops from, or formed at, absorbed and absorbing vertices amounts to a context-driven decision about the absorption process that is unlikely to be of any utility and need not be considered. Therefore, we proceed here to consider the space of such possible fixed choices. In the context of control flow graphs, a loop corresponds closely to a do-while construct. With this in mind, preserving such a construction under absorption would correspond to inserting additional computations into a do-while loop or forming a new do-while loop around existing computations, altering the control flow. Meanwhile, annihilating loops would correspond to embedding the do-while construct within a larger sequence of computations, preserving the control flow. This is *prima facie* cause to restrict consideration to the definition of absorption introduced above.
- [40] Note that we are not explicitly considering the insertion of loops in this setting.
- [41] In particular, loops and reflexive self-edges are not included here, though the former may be accommodated without substantial changes.
- [42] See Definition 1.4.8 of [16].
- [43] NB. A notion of “flow” and a concomitant ω -categorical framework is discussed in [30]. This construction is primarily concerned with a space of paths for modeling concurrency and connections to the present work are not obvious.
- [44] See Definition 1.4.5 of [16].
- [45] These can be respectively called from MATLAB along the lines of <https://github.com/rogalski/triconnected-components> and <http://www.mathworks.com/matlabcentral/answers/30377>.
- [46] An alternative algorithm has been put forward by [31] but no implementation of it is known to exist.
- [47] For acyclic TTGs it is not hard to see that the analogue of SESE regions are vertex pairs (j, k) s.t. $D_{jk}D_{jk}^\dagger - \delta_{jk} = 1$, but the cyclic case is much harder.